A SERVICE ORIENTED ARCHITECTURE FOR ROBOTIC PLATFORMS

BY

PAUL D. HESTAND
B.S., DAVID LIPSCOMB UNIVERSITY (1983)
B.S., THE PENNSYLVANIA STATE UNIVERSITY (1986)
M.S., UNIVERSITY OF MASSACHUSETTS LOWELL (1999)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL

Signature of Author:_____ **Date:**_____

Signature of Dissertation Chair:_____

Prof. Holly A. Yanco

Signatures of Other Dissertation Committee Members

Signature of Committee Member:_____

Dr. Stephen Balakirsky

Signature of Committee Member:_____

Prof. William Moloney

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
MAR 2011		**00-00-2011 to 00-00-2011**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Service Oriented Architecture for Robotic Platforms	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Massachusetts Lowell,One University Avenue ,Lowell,MA,01854**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Existing architectures, infrastructures, and frameworks for robotic systems have limited utility when used in the context of integration. This limited utility derives from either a narrow scope assigned to the architecture or a lack of application of generally accepted software engineering principles for integration. Commercial development strategies utilizing well-de ned software architectural principles are frequently at odds with how typical robotic software architectures are designed. This inherent con ict results in resource consumption on integration work in projects where integration is not the primary goal. Moving results out of the laboratory and into the commercial domain becomes di cult. In this thesis, I present an architecture based on a clear de nition of software architecture, an understanding of the stakeholders for the architecture, those stakeholder interests, and the use of accepted principles of software architecture definition and design. A software architecture is constructed around the viewpoint that a robotic system can be considered an enterprise and everything associated with that enterprise provides a service. This enterprise perspective leads to the description of a service oriented architecture (SOA), which provides integration exibility while satisfying stakeholder requirements and interests. I construct an implementation approach embodying these principles and apply that implementation approach to real world integration problems to illustrate the utility of such an approach in robotic software development. Finally, I de ne a set of metrics to be used when comparing the SOA approach with other architectures for integration on robotic systems.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**227**	

A SERVICE ORIENTED ARCHITECTURE FOR ROBOTIC PLATFORMS

BY
PAUL D. HESTAND

ABSTRACT OF A DISSERTATION SUBMITTED TO THE FACULTY OF THE
DEPARTMENT OF COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY IN
COMPUTER SCIENCE
UNIVERSITY OF MASSACHUSETTS LOWELL
2011

Dissertation Supervisor: Prof. Holly A. Yanco
Associate Professor,
Department of Computer Science

ABSTRACT

Existing architectures, infrastructures, and frameworks for robotic systems have limited utility when used in the context of integration. This limited utility derives from either a narrow scope assigned to the architecture or a lack of application of generally accepted software engineering principles for integration. Commercial development strategies utilizing well-defined software architectural principles are frequently at odds with how typical robotic software architectures are designed. This inherent conflict results in resource consumption on integration work in projects where integration is not the primary goal. Moving results out of the laboratory and into the commercial domain becomes difficult.

In this thesis, I present an architecture based on a clear definition of software architecture, an understanding of the stakeholders for the architecture, those stakeholder interests, and the use of accepted principles of software architecture definition and design. A software architecture is constructed around the viewpoint that a robotic system can be considered an enterprise and everything associated with that enterprise provides a service. This enterprise perspective leads to the description of a service oriented architecture (SOA), which provides integration flexibility while satisfying stakeholder requirements and interests. I construct an implementation approach embodying these principles and apply that implementation approach to real world integration problems to illustrate the utility of such an approach in robotic software development. Finally, I define a set of metrics to be used when comparing the SOA approach with other architectures for integration on robotic systems.

ACKNOWLEDGEMENTS

This research was funded in part by an Army Research Office MURI (W911NF-07-1-0216).

I would like to thank Dr. Holly Yanco for taking me on as her student. She has been inspirational in her passion for research with integrity and in her insistent nudging me towards completion. I would also like to thank the members of my committee, Prof. Bill Moloney and Dr. Steve Balakirsky, for their insightful comments and discussions. The Computer Science Department at The University of Massachusetts Lowell deserves a special thank you for providing an encouraging and thoughtful environment in which I could maintain my fulltime job and still work to complete the degree. Thank you especially to Dr. Jim Canning, Dr. Tom Costello, and Dr. Jie Wang as the Department Chairs for leading that. I would also like to thank the various employers and customers whom I've had the privilege to work with over the course of this research. The opportunities to hone my understanding of the practice of software engineering in their various work environments with emphasis on very different aspects of software engineering and architecture have made me a much better architect and computer scientist through connecting the theory with real world problems. Of all the lessons I've learned through the commercial practice, the most important is that software engineering and software architecture are much more interesting when applied in a production environment and especially when things go badly and you are forced to reevaluate why and how to do it better the next time. Thank you also to my colleagues in the Robotics Lab for the many interesting discussions and general atmosphere of collaboration. I especially want to give my deepest gratitude to my family; my wife, Barbara, and daughters, Kristen and Mallory who suffered through my seemingly eternal student status. Without their support and understanding this would not have been possible.

I would like to dedicate this thesis and the work behind it to Dr. Ray Gumb and Prof. Charlie Steele who as former advisors were instrumental in my enthusiasm for academic computer science and for my decision to continue on to this degree and to persevere. They are sorely missed but never forgotten.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

Part I

Problem Statement and Background

1. PROBLEM STATEMENT

Integration of new hardware and software components in robotic systems consumes significant effort and budget in projects for which integration is not the primary focus. Yet software and hardware integration is a fundamental activity associated with the research, development, and maintenance of robotic systems. Commercial software development organizations, particularly those organizations associated with business enterprises, have standardized architectural strategies and principles focused specifically on facilitating integration. In addition, these commercial organizations apply these same strategies and principles as a part of the internal engineering culture. However, current practices in robotics software development make limited or no use of these accepted software architectural practices. Failure or reluctance to consistently adopt these accepted practices within the robotics community results in systems that are unnecessarily constrained, difficult to maintain or extend, and applicable to a single platform, a limited group of platforms, or a single aspect of the robotic platform. The problem is further compounded when an existing system is adapted to an unsuitable architecture without consideration for sound software architectural principles.

Numerous attempts have been made to correct these problems (e.g., Côté et al. [2004], Chaimowicz et al. [2003], Kenn et al. [2003], Colon et al. [2006], Kapoor [1996], Volpe et al. [2000], Gorton and Mikhak [2004], Hulin [2003], Gertz [1993]). However, these attempts have not gone far enough because of self-imposed scope limits or because the corrections are a "duct-tape" integration of existing systems without consideration for the overall architectural goals of the new system. The latter is in fact not a correction to the problem but a different manifestation of the same problem. In addition, the differences in requirements and quality attribute valuation between commercial, research, and military systems make integration goals a moving

(if not missing) target. This lack of clearly defined integration goals when coupled with an extensive number of API, infrastructure, programming model, framework, and architecture choices leave the roboticist with a large integration effort or a new development effort, both of which are costly and may result in a solution that is marginally satisfactory. Additionally, the ability to rapidly transition new capabilities from the laboratory to practical use is hindered because of an inability to quickly and easily integrate those new capabilities with existing systems.

In this thesis, I present an integration approach based on the use of best practice software architecture and engineering principles along with concepts from the "Enterprise Computing" domain[1] and component-based design principles. The goal of this approach is to create an integration environment in which flexibility and extensibility are of primary importance for implementing robotic platform solutions. In addition, I have created a prototype reference implementation based on the architectural approach to validate the principles and provide a basis for comparison with other integration approaches currently used in robotics. The comparison is made in the context of a set of metrics chosen to characterize the difficulty or ease of doing integration. Finally, I present the results of application of the architectural approach to different types of integration activities for a single robotic platform and show how the metrics computed for that application compare to other approaches used for the same activities. Thde integration activities are not "toy" problems but represent real integration tasks from a Multi-University Research Initiative (MURI). Following analysis and discussion of the results, I will present the research contributions and limitations of the work presented here and identify extensions to the approach which can be used to move the reference implementation from a proof-of-concept prototype to a more robust and operational framework.

[1]Enterprise computing is typically used to discuss business systems that must interact but may exist on heterogeneous systems usually as a result of legacy systems or merger and acquisition activities.

2. SOFTWARE ARCHITECTURE BACKGROUND

Understanding the benefit of using an architecture to enable integration first requires knowledge of what a software architecture represents. Consideration of existing architectures designed for integration purposes and of the use of architectures in robotics will further aid in formulating an approach for creation of an integration architecture for use in robotics. In this chapter, we examine architecture from a software engineering perspective and present design considerations for the proper design of software architectures.

2.1 Architectures, Infrastructures, and Frameworks

2.1.1 DEFINITIONS AND TERMINOLOGY

There exist three concepts that are used interchangeably in discussions of architecture. Those concepts are architectures, infrastructures, and frameworks. However, the three terms represent different structures and are not truly interchangeable. Thus, it is important to understand the differences between these concepts and how those differences affect the ability to do integration effectively.

We start with the definition of a software architecture. An good working definition is given by the Software Engineering Institute in their work on formalizing architectural principles and practice. This definition has also been adopted as a standard definition in IEEE Standard 1471-2000 [IEEE, 2000]. That definition is

2.1.1 Definition

Architecture "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [p.3]

We extend this definition by adding that an architecture also embodies the technical decisions made in support of business goals since software architectures are created, used, and evolve to support the successful operation of the business owning the architecture. Organizations not making explicit architectural decisions, i.e., just developing code to get to market, have made architectural decisions in the way that the "just developing code" creates some form of usable product consisting of elements in relation to each other. Architectures formed this way tend to be difficult to modify, extend, or maintain because these characteristics were not considered when developing the code. The identification of the business goals that drive technical decisions is important for robotics, too. To understand how software architectures can be and are used in robotics, an understanding of the business goals, which might be more properly called "research goals" or "mission goals," helps in making technical and tradeoff decisions which are driven by the requirements of research or the requirements defined by the intended use of the robotic system.

The definition of infrastructure is "the underlying foundation or basic framework (as of a system or organization)."[1] A software infrastructure provides the foundational elements that support the creation and use of software following a particular model. An example of an infrastructure is CORBA[2], a middleware infrastructure enabling more efficient development and use of distributed applications. Note that an infrastructure is more concrete than an architecture and more focused. An architecture might be realized through an infrastructure or a collection of infrastructures.

A framework is defined as "a basic conceptional structure."[3] Wikipedia defines a software framework as "...an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality..."[4] Thus, a framework is a set of constructs created to support software development following an explicit set of rules. Typical frameworks include graphics frameworks such as OpenGL and DirectX, system frameworks such

[1] *Merriam-Webster's Collegiate Dictionary, 11th Edition*, s.v. "infrastructure"
[2] Common Object Request Broker Architecture
[3] *Merriam-Webster's Collegiate Dictionary, 11th Edition*, s.v. "framework"
[4] *http://en.wikipedia.org/wiki/Software_Framework*

as Portable Operating System Interface (POSIX) threads, and distributed system frameworks such as Java 2, Enterprise Edition (J2EE). All of these are implementations providing code constructs that enable developers to follow well-defined sets of coding rules. J2EE is slightly different in that it might also be regarded as an infrastructure, but there exist framework elements within the J2EE specification. Again, note that a framework provides less abstraction and is more tightly focused than an infrastructure. A final example relating specifically to architecture is *The Open Group Architecture Framework™* (TOGAF) [The Open Group, 2010a] which is a framework specification for enterprise architectures. However, TOGAF™ is different from the previously described frameworks in the inclusion of specifying how architectures should be developed, what content should be included, and descriptions of reference architectures. It is worth noting that the TOGAF™ definition of an architecture was extended to include the description of the architecture as an acceptable definition of architecture[The Open Group, 2010b]. We believe, based on experience, that this extension leads to an inconsistent and confusing use of the term "architecture" when used without proper context. To maintain clarity, we will not use the description of an architecture as the architecture itself but simply as one representation of the architecture. Other representations of an architecture exist to include a system implementation based on the architecture or a reference architecture description used to show usage, guide development, and provide conformance criteria.

What can be understood from all of this is that an architecture is neither an infrastructure nor a framework. It is more, incorporating possibly many infrastructures and frameworks and defining how those elements[5] are related to each other, both statically and dynamically. It is this definition of an architecture that will guide our development of an integration architecture for robotic platforms. The architecture must provide the externally visible properties of those elements (i.e., their interfaces), define how they interact, and define how they are to be used within the architecture. The usage model may be done via a reference architecture, a reference

[5]We use the more general term element instead of component so as to distinguish between things that make up an architecture and the specific use in the term "component architectures."

implementation, or other similar implementation.

2.1.2 ARCHITECTURE EXAMPLES

There are a large variety of different architectures, reference architectures, and architectural frameworks. Enterprise architecture is an architectural category that is used to describe the elements, structures, processes, and relations of enterprises. The term "enterprise architecture" does not refer specifically to software architecture but has come to be associated with that use. This association is a result of the alignment of the Information Technology (IT) functions with the enterprise architecture as a means of enablement. Of the many different ways of describing enterprise architectures, the most commonly used is called the Zachman Framework, defined by John Zachman [Sowa and Zachman, 1992, Zachman, 2010]. An early application of the Zachman Framework is demonstrated in [Dedene and Snoeck, 1994]. The Zachman Framework is not specific to software architectural description but provides a basis for many of the enterprise software architectural descriptions including TOGAF™ [The Open Group, 2010a], The Treasury Enterprise Architecture Framework [US Dept of the Treasury CIO Council, 2001], and The Department of Defense Architecture Framework (DoDAF) [DoD Deputy CIO, 2009].

Reference architectures provide a standard against which other architectures may be evaluated for conformance to a particular architecture model. Examples of reference architectures include the OASIS SOA Reference Architecture [OASIS, 2009], the Federal Enterprise Architecture (FEA) [U.S. Office of Management and Budget, 2010], and the DoD Business Enterprise Architecture (BEA) [DoD Business Transformation Agency, 2010]. Within the robotics community, two notable reference architectures are the 4-Dimensional/Real-time Control System (4D/RCS) architecture [Albus, 2002] and the Joint Architecture for Unmanned Systems (JAUS) [Department of Defense, 2004]. We will return to 4D/RCS and JAUS in Chapter 3. For our service-oriented approach, we provide a *de facto* reference architecture through documenting the expected structure constituting a service-oriented architecture, expected patterns of interaction, and through application of service-oriented principles

to specific integration tasks to illustrate usage and provide a basis for measurement.

2.2 Architecture Principles and Patterns

Software architecture principles are guidelines for applying architecture [The Open Group, 2010a]. The principles provide guidance on what constitutes a satisfactory architecture and give reference points on identifying when an architecture is unsuitable. Architecture patterns, on the other hand, are applications of architectural principles and software components in such a way that similar problems can be repeatedly solved using the same or equivalent combinations. In this way, architecture patterns are related to software design patterns as described in [Gamma et al., 1993, 1994]. In fact, some architectural patterns are just generalizations of software design patterns described in Gamma et al. [1994].

Architectural principles for software have been around since at least 1962 when Fred Brooks is credited with having coined the term "architecture" [Buchholz, 1962, Clements et al., 2002]. The focus of architectural principles has been on different things at different times. Some focus areas include hardware, user interface, or even parts of the software visible to the user (not necessarily the interface). Recently, however, the concepts of architecture, what architecture is, and the principles underlying software architecture have begun to be codified through standardized definitions and standards-based architectural principles, primarily through work done at the Software Engineering Institute (SEI) at Carnegie Mellon University [SEI, 2008], but also through The Open Group in their work on TOGAF™ [The Open Group, 2010a], the Organization for the Advancement of Structured Information Standards (OASIS) [OASIS, 2010], and the United States Office of Management and Budget in specification of the Federal Enterprise Architecture (FEA) [U.S. Office of Management and Budget, 2010].

In the next sections, we will discuss architectural principles and patterns and how they can be used in the design of robotic or autonomous platform software architectures.

2.2.1 ARCHITECTURAL PRINCIPLES

As mentioned earlier, architectural principles are guidelines for applying software architecture. TOGAF™ identifies common architectural principles and phrases those principles in business and information technology terminology with the aim of applicability within the business enterprise context. We state a subset of those principles that have applicability to autonomous systems here but recast them in more general terms for alignment with autonomous systems.[6] The applicable principles are

1. Primacy of Principles—the enterprise can only succeed when all participants follow the same rules,

2. Maximize Benefit to the Enterprise—decisions must be made in the context of the larger enterprise, not in a selfish fashion for the benefit of a single organization or function,

3. Business Continuity—the enterprise must continue to function in spite of interruptions,

4. Common Use Applications—application development must be applicable (as much as possible) to the organization as a whole and not to any individual organization or group,

5. Service Orientation—the enterprise is organized around delivery and consumption of services representing the processes used in interaction between members of the enterprise,

6. IT Responsibility—the infrastructure is responsible for ensuring that the (reasonable) needs of all enterprise participants are equitably met and for identifying the conditions for how and why the delivery expectation may degrade, and

[6] For a detailed explanation and additional principles cf. The Open Group [2010a]. The principles cited here are those found in TOGAF™ having the most direct applicability to autonomous systems when defining integration and architectures to support integration.

7. Protection of Intellectual Property—the enterprise is heavily invested in intellectual property and the infrastructure must ensure the reasonable protection of that investment.

Each architect ultimately must decide on an appropriate set of applicable architectural principles. The principles for a particular architecture are derived from various sources including the type of architecture, the existing and planned applications supported by the architecture, organizational culture, goals, constraints and processes, laws and regulations, or they may be inherited from the use of a reference architecture or reference model. For application in autonomous systems, the enterprise may take various forms including the collection of all subsystems on a robotic platform, a group of collaborating robots, and multiple, collaborating groups of robots. We will to return these perspectives in later chapters but it is important to understand that the expectations for a business enterprise do not differ greatly from those for a robotic system and this parallel is the basis for how service orientation applies to integration for robotics.

2.2.2 ARCHITECTURAL PATTERNS

TOGAF™ defines architectural patterns as [The Open Group, 2010a, page 1, Architecture Patterns]

> "In TOGAF, patterns are considered to be a way of putting building blocks into context; for example, to describe a re-usable solution to a problem. Building blocks are what you use: patterns can tell you how you use them, when, why, and what trade-offs you have to make in doing so."

Software design patterns are familiar to anyone who has read the "Gang of Four" Design Patterns book [Gamma et al., 1994] or spent much time browsing the web for current practices in software development. Patterns such as *proxy* and *mediator* are so common that to cite examples could fill several pages. The utility of these patterns lies in their descriptive nature which permits a rapid means of communicating exactly what is intended for a particular design. Architectural patterns have a similar communicative ability but have not been in such widespread use. Different architecture

patterns have been identified and collected into various repositories or described in books on patterns (e.g., Erl [2010, 2009b], Booch [2010], The Hillside Group [2010], Fowler [2003], Hohpe and Woolf [2004, 2003]. The description of patterns used by TOGAF™ is based on the IBM Redbook Series [Galic et al., 2003] and classifies architecture patterns hierarchically as

1. Business patterns—identify business actors and describe the interactions between them in terms of typical business interactions,

2. Integration patterns—used to combine business patterns into a "solution." These are typically described as workflows or business processes,

3. Composite patterns—identify combinations of business and integration patterns such as those used in eCommerce applications,

4. Application patterns—implementation patterns used for each business and integration pattern, and

5. Runtime patterns—implement certain quality attributes or "service level characteristics" such as performance or scalability.

Each pattern group is typically used in enterprise business systems. We believe they also have applicability in the architecture of a robotic software system. For instance, *Business patterns* include the *Collaboration pattern* in which users work with one another to share data and information. This type of collaboration is present in an autonomous system in which the users can be viewed as components/agents in the autonomous system sharing sensor data or mission planning information with other elements to create a functioning robot. We discuss this further in Chapter 5 with particular focus on the use of the *Service pattern*.

An important set of architectural patterns are found in the three broad categories of patterns identified as

1. Layered—elements are grouped into layers that are arranged hierarchically based on some form of process flow or level of detail. Strict layering requires that communication only occur between adjacent layers using interfaces,

2. Pipeline—elements are grouped into a sequenced ordering typically associated with the flow of data or control through some process. Another term for this pattern is "pipe-and-filter" to indicate that the flow is through pipes and the elements act as filters on whatever is flowing, and

3. Messaging—elements are loosely arranged and communication occurs through the sending and receipt of messages between the elements. In many messaging architectures, a central collection of elements may serve as a dispatch and routing hub to further isolate elements from each other. Such a hub is called a "message broker."

We will use these categories of architectural patterns in Chapter 3 to classify existing robotic architectures.

2.3 Architecture Documentation

Designing a software architecture entails more than just making decisions about the choices of components and how to connect those components. Those decisions and their consequences have to be clearly communicated to anyone with an interest in the architecture. As stated in Clements et al. [2004],

> "The perfect architecture is useless if it has not been expressed understandably."

Thus, describing the architecture accurately and comprehensibly becomes an important task.

There have been many approaches defined for the description and documentation of software architectures. As mentioned previously, architecture frameworks such as TOGAF™ define how architecture descriptions should be made and what artifacts should be created. Formal description languages termed *Architecture Description Languages*, or ADLs, have been designed to describe architectures in more precise terms using mathematical or other formalisms related to modeling. In Medvidovic and Taylor [2000], several ADLs are identified and compared, all taking a

different perspective on the architecture but all based on a formal specification of the architecture.

Other description approaches include the "4+1" view model as described in Kruchten [1995]. A *view* is defined as a representation of a set of elements and the relationships between them. The "4+1" approach identifies a set of four views of the architecture including logical, process, development, and physical. These four views provide a description of the architecture appropriate to different groups interacting with the architecture at various stages in the lifecycle of the architecture. The "+1" view is a set of scenarios associated with each view and showing the connection between the other four views. This approach has seen widespread acceptance in commercial software development organizations and is well supported by commonly used modeling tools.

Researchers at the SEI have extended the view concept to provide a richer description capability. This view-based approach is described in Clements et al. [2004] and is called "Views and Beyond". The "Views and Beyond" approach provides a representation and documentation toolset from which an architectural description can be created and made comprehensible to most audiences. We will base the description of our architecture on the "Views and Beyond" approach and will refer to "Views and Beyond" as simply *V&B*. The remainder of this section will describe the V&B approach in more detail.

The V&B approach to documentation defines a taxonomy of views from which an architect selects to represent the architecture. The documentation structure associated with those choices then refines the perspectives and content the architect should provide. The first level of view decomposition is termed *viewtypes* and consists of

1. Module—fundamental unit of implementation in the system and composed of different elements and relationships. A module, in theory, represents an independent element with no external dependencies,

2. Component-and-Connector (C&C)—a fundamental unit of execution in the system describing components and how those components are connected to each

other, and

3. Allocation—the relationships between the system's software elements and the development and execution environments in which those elements reside. Allocation viewpoints are typically used to associate specific architectural elements with physical elements such as software files, libraries, processes, or hardware.

Each viewtype is further decomposed into *styles*. Styles are architecture families satisfying a given set of constraints and solving certain types of problems in a more or less uniform, technology independent way. Architecture styles are related to architecture patterns through the association of pattern groups with certain styles. Views are associated with styles and represent application of a style to a particular system. There exist many different architecture styles but the more commonly used styles are [Clements et al., 2004]

1. Module viewtype
   (a) Decomposition—shows "is-part-of" relationships between modules and elements making up the module,
   (b) Uses—shows dependency relationships between modules,
   (c) Generalization—shows extension or inheritance relationships between modules, and
   (d) Layered—shows system partitioning into distinct layers with ordering relations and interaction constraints on the layers.
2. C&C viewtype
   (a) Pipe and Filter—shows data transformations as a flow of data through a series of interconnected filters where each filter applies some transformation to the data,
   (b) Shared Data—shows relationships between data, persistent storage of data, and users of that data,

(c) Publish-Subscribe—shows interaction between components as events published by a component and subscribed to by other interested components,

(d) Client-Server—shows interaction between components as requests for service by a client and responses to those requests by a server or servers,

(e) Peer-to-Peer—shows interaction between components as sequences of client-server type interactions with client and server roles being exchanged at each interaction, and

(f) Communicating Processes—shows interactions between components executing concurrently and describes behaviors specific to concurrency.

3. Allocation viewtype

(a) Deployment—shows allocation of C&C elements to execution environments including physical hardware,

(b) Implementation—shows allocation of the development work products to the development infrastructure, and

(c) Work Assignment—shows allocation of system elements to development resources

Additional documentation types that are valuable for understanding an architecture but which do not fit within either a viewtype or style include

1. Context—shows the relationship of the system to the environment external to the system including external entities using the system and external entities affected by the system operation. DoDAF v.2documentation describes context in a graphical representation [DoD Deputy CIO, 2009],

2. Combined Views—useful when one or more elements appears in multiple views to show how the multiple views are similar or represent transformations of each other,

3. Variability—shows how the architecture can be used to create different systems from the same base architecture through variation of architectural elements, re-

lationships, or behaviors. COVAMOF is an example of a variability description framework [Sinnema et al., 2004],

4. Dynamism—describes architectures that may change structure or form during execution of the system,

5. Interfaces—describes the contractual relationship between providers of functionality and users of that functionality as expressed through the public interfaces,

6. Behavior—describes different forms of communication, constraints on ordering of items like messages, or behaviors associated with real-time modes of operation such as clock-triggered events, and

7. Requirements—describes capabilities the system must provide (functional) and characteristics the system must exhibit (non-functional).

Architecture views are used for communicating the design, construction, and functioning of the architecture to all stakeholders associated with the system. The stakeholders are those people, groups, or systems impacted in either a positive or negative way by the architecture design. The availability of view choices is quite large and it is up to the architect to select a set of views appropriate to the architecture that provides the greatest benefit to each stakeholder. This selection requires an analysis of the different stakeholders, their motivation as a stakeholder, and the type of architecture description best meeting their needs. We will return to this when we define the requirements for an architecture that is designed with integration as the primary goal.

In the next chapter, we consider the types of architectures and frameworks in use within the robotics community. In addition, we will discuss how each architecture or framework supports integration.

3. ROBOTIC ARCHITECTURES BACKGROUND

As a means of motivating our design of an integration architecture for robotics, we will examine recent work on architectures, infrastructures, and frameworks for robotics and consider the support for or limitations to integration.

3.1 Robotic Architectures

In addition to software architectures as described in Chapter 2, the term "architecture" is used in robotics to refer to models of how actions or control are associated with sensing. These robotic architectures are similar to software architectures but provide descriptions of how robotic components generate observed behaviors of the robot. Three classes of robotic architectures are generally recognized [Murphy, 2000]:

1. Deliberative or Cognitive
2. Reactive
3. Hybrid

The Deliberative architecture was formulated to allow sensors to provide input to a cognitive module for generation of plans or schedules for the next action or actions to be taken by the robot. The quintessential example of a deliberative architecture is that found in Shakey the Robot[1] [Nilsson, 1984]. In Deliberative architectures, sensors provide data to a planner or scheduler which uses the sensor input to define a set of control commands sent to actuators to create an action. No action is taken without direction from the planner and all sensor outputs are fed into the planner.

[1]The work on Shakey started in the mid-1960s at Stanford Research Institute and continued for over a decade. A good description and the technical reports on Shakey may be found at http://www.ai.sri.com/shakey/.

Deliberative architectures have limitations associated with how fast the robot can respond to stimulus because such a response must be generated by the planner and if the planner is slow in delivering control commands, the robot will appear to be unresponsive or very slow to respond. Shakey's movement was very shaky since it could not move very far until the next set of commands were provided by the planner.

Reactive architectures were first described by Brooks [Brooks, 1986] and Arkin [Arkin, 1989] for mobile robots. These architectures are primarily recognized by the absence of any planning ability. Thus, in a reactive architecture, an actuator or control responds directly to a stimulus obtained from an input sensor. The overhead associated with planning is avoided by creation of reactive behaviors that are immediate and based strictly on the sensor's input. This behavior is similar to that exhibited by cockroaches when they flee in response to light or in the typical human reaction when touching a hot item. Reactive architectures can exhibit very rapid response times to a stimulus but only if the appropriate behavior is provided for a particular association between sensor and actuator or control.

The Hybrid architecture blends features of both reactive and deliberative architectures. In a hybrid architecture, a planner is responsible for coordinating the use of certain types of behaviors based on sensory input while other behaviors are provided in a reactive manner. A robot employing a hybrid architecture could be given a mission that requires the platform to travel to some set of locations, searching for something along the way, and mapping the areas explored while reacting to certain dangerous conditions without requiring planning input.

Robotic architectures can be considered as system (software, hardware, and processes) architectures with the primary focus on achieving behavioral goals, e.g., searching for items, reacting to a stimulus, conducting a surveillance mission in a defined geographic area. This architectural focus is not necessarily concerned with the overall software architecture requirements but drives a particular approach to the structure and organization of the software elements, i.e., the software architecture. Any software architecture proposed for use in integration for robotic plaforms must account for the inclusion of these robotic architectures. In the next sections, we will

examine existing architectures for robotic applications and show how they map into software architectural patterns or models.

3.1.1 RECENT WORK IN ROBOTIC SOFTWARE DESIGN

3.1.1.1 4D/RCS

The 4D/RCS[2] is an architecture developed by James Albus at NIST [Albus, 2002] for specifying the control of robotic vehicle systems. It incorporates elements of both reactive and deliberative architectures and is structured like a military chain of command with different levels of command and control. Each level represents a planning level with a well-defined time horizon. The time horizon increases as one moves up the chain-of-command from real sensors and actuators towards the battalion level. At each level are a set of computation nodes incorporating world maps, behavior generators, and sensory processing. The architecture maintains a world map which is updated via sensor and actuator proxies at each level. This information is collated and passed up the chain of command while commanding instructions are passed down from the appropriate level to the sensors and actuators. This architecture is limited in that it only allows a hierarchical control structure, which limits the flexibility of the system such that when new sensors or actuators are added to the system, knowledge of their existence must be propagated to all levels of the control structure for effective use.

The hierarchical, layering structure of 4D/RCS is a common software architectural pattern. The layer pattern is intended to enforce strict flow of control and data only between adjacent layers. The difficulty in most layered architectures is that because the coupling between adjacent layers is typically underspecified, control and data bleed through layers due to direct calling into non-adjacent layers. These direct calls may be required to access data unavailable in adjacent layers via the layer interfaces. Such "layer bridging" or "cross layering" works, but promotes strong coupling in non-adjacent layers and makes the architecture difficult to understand, maintain, and

[2] 4-Dimensional/Real-time Control System

evolve (cf., Kawadia and Kumar [2005] for a fuller explanation of the difficulties). We believe 4D/RCS is a good architecture to incorporate into a much larger architectural specification since it is aimed specifically at vehicle systems and focused on real-time control of these systems.

3.1.1.2 Subsumption

Subsumption [Brooks, 1986] is the name given to a particular type of reactive system proposed and implemented by Rodney Brooks at the MIT Artificial Intelligence Laboratory. Subsumption is based on viewing the behavior of the robot in terms of layers of competency. The name "subsumption" comes from the fact that in this multi-layered architecture, higher layers may subsume the behaviors of lower levels [Brooks, 1986]. Complex behaviors are built up from combining simple atomic behaviors such as move, sense, and run away. The more complex behaviors subsume the simpler behaviors by taking control away from the lower layers when indicated by some sensory or control input.

Subsumption as a software architecture makes use of layering to achieve the goals of additivity and robustness [Brooks, 1986]. These goals are met by encapsulating reactive behavior in the lowest layers and simply replacing the atomic behavior when needed, such as when a sensor fails, for instance. The fact that Subsumption uses layering as the architectural pattern presents the same difficulties described in the Section 3.1.1.1.

3.1.1.3 3-Tier Architecture (3T)

The 3T architecture is a control architecture first proposed by Bonasso et al. [Bonasso et al., 1995] in 1995. 3T architectures focus on control of a system through three separate, communicating layers. The first layer is the "reactive skills" layer, in which a set of reactive behaviors are maintained along with a skill manager. The skill manager is responsible for activating or terminating specific skills based on sensor input or inputs from the higher layers. The second layer is the "sequencing" layer. This layer is responsible for decomposing a plan provided by a higher layer

into a set of skill requirements for activation in the reactive layer. The third layer is the "deliberative" layer and is responsible for formulating plans based on mission requirements. The formulation is not really an ad hoc formulation but a list of plans corresponding to specific mission level taskings.

The 3T architecture is a software architecture used to specify how control is effected within a robotic system. As such, it is focused on promoting flexibility and robustness. Flexibility is required to accommodate new skills and plans while robustness provides the ability to continue functioning in the presence of failure. Given that the 3T architecture was designed to deal with control of autonomous systems in space environments, these requirements are very important.[3] The tiers in 3T represent a layered architecture although in some implementations of 3T, messaging is used between the different tiers.

3.1.1.4 Player

Player [Gerkey et al., 2001] is a distributed control protocol based on socket communications developed by the University of Southern California Robotics Research Laboratory. The goal is to provide transparent access to sensors and controls of robots in an environment using POSIX socket constructs. This definition is useful since most programming languages and operating systems support sockets in a mostly uniform fashion. As described in the original paper, Player is a protocol description and can be implemented by any software implementing that protocol. The Player protocol may also be used in a sensor and control simulation environment called Stage [Gerkey et al., 2003] and in Gazebo for simulating multiple robots [Vaughan et al., 2003].

The difficulty with Player is that while sockets are ubiquitous, they do not permit ready control from truly remote clients, e.g., a geographically dislocated client. Further, while Player specifies a standard protocol, it is not sufficient for Player to

[3] We should be careful not to confuse the 3T architecture in robotics from a 3-tier architecture as used in software engineering. In the form used in software architecture, the tiers represent user interface, functional processing, and data access. There are similarities but the correspondence is not exact and the goals are different.

be considered an architecture. We believe that despite these shortcomings, Player is a very usable protocol and easily fits into a larger software architecure description. In later chapters, we will show one example of Player used as the foundation for an Enterprise Service Bus; a role which it easily fulfills.

3.1.1.5 JAUS

JAUS[4] is an architecture defined by the U.S. Department of Defense (DoD) and intended for use in all autonomous systems developed for DoD use. Its chief characteristic is a component-based messaging system. The organization of the components follows a hierarchical structure with "commanders" at different levels for controlling the flow of data and control both up and down the chain of command. The specification of JAUS consists of a description of the domain of applicability for JAUS, a description of the components expected in a JAUS compliant implementation, message definitions, and a compliance guide for determining whether a particular implementation complies with the requirements of JAUS [Department of Defense, 2004]. As of 2005, the JAUS specification was taken over by the Society of Automotive Engineers (SAE) as the AS-4 standard [Department of Defense, 2005]. The current version of JAUS is 3.3 but version 4.0 is in draft at the time of writing of this thesis and represents a significant change over the previous versions approach to unmanned systems architecture.

While JAUS is intended to specify interoperability, it constrains the components to use strict messaging with pre-defined messages for interaction between components. Strict messaging is useful in many situations but may limit integration of components that are not message-capable, as may be the case for legacy components. In addition, JAUS does not specify how components such as hardware abstraction, display, and data integration are incorporated in the overall system. We believe that JAUS may be suitable for command and control architectures but is inadequate for broader uses and may not even be usable without modification for systems incorporating legacy software components.

[4] Joint Architecture for Unmanned Systems

3.1.1.6 Joint Technical Architecture

The Joint Technical Architecture [JTADG, 2003], or JTA, is a Department of Defense (DoD) standard specifying three views of an interoperable system architecture. These views are operational, technical, and system. The JTA specifies subdomains focusing on concepts such as cryptography, ground systems, aerial systems, and the like. The intent of the JTA is to provide a unifying view on integrating widely disparate systems such as communications, warfighting machinery, and people in such a way that DoD contractors and their systems can more easily interoperate.

Since the JTA's intent is more broad than just autonomous systems, it is an abstract system architectural specification for large systems. As such, its applicability is limited when used to specify the software interoperability of autonomous systems. However, JAUS was created as a JTA-compliant architecture, and so the guidance provided by the JTA is useful for defining more concrete architectures with direct applicability to autonomous systems.

3.1.1.7 Pyro

Pyro [Blank et al., 2003, 2004, 2006] stands for "Python Robotics" and is a programming environment intended to be used in a lab setting. The base programming language is Python, which is an object-oriented interpreted language. The interpreted capability provides a means for quickly turning around code changes to check out the effect. As an environment, it also provides a set of base objects which provide a level of abstraction above the hardware. This abstraction means that the user doesn't have to worry (too much) about the details of the hardware and can focus on the software interaction of the system objects. Since Python also provides an ability to compile the source code, performance can be optimized (within the limits of the language) for use outside of the lab.

This system is targeted to a single language as implied by its name. This limitation may not be too severe if one is interested in only using Python. Tools exist to convert C and C++ to Python such as the Simplified Wrapper and Interface

Generator (SWIG) [Beazley, 2007]. However, in a commercial environment, most engineering shops are reluctant to develop in one language and transform to another due to maintainability issues. In safety-centric systems such as those used in medical environments or those requiring Federal Aviation Administration (FAA) certification, it may not be possible to guarantee the safety of transformed code. Additionally, most large-scale systems require integration of components that may not be amenable to use with or within Python or may exist in languages such as Matlab, Fortran, or Assembler for which no transformation to Python is available. Furthermore, the facilities for integration are limited to whatever can be done with the existing base components or require writing new components. Thus, integration can potentially be timely and costly.

3.1.1.8 MARIE

MARIE [Côté et al., 2004] stands for Mobile and Autonomous Robotic Integration Environment. MARIE is being developed with the purpose of accelerating software integration for robotic platforms. The basis of MARIE is a software design pattern known as the MEDIATOR pattern [Gamma et al., 1993]. This pattern creates loose coupling of distributed software components by designating a centralized control component known as the "mediator" which controls and coordinates "peer" components. The loose coupling permits integration of heterogeneous components with little regard for development language, communications protocol, or underlying execution mechanisms. In addition, MARIE is designed as a layered architecture with layers supporting core abstractions, component creation and management, and application building.

MARIE answers many questions related to the integration of components used in robotic or autonomous applications. However, the layering structure forces decisions about how components are created and managed and what things are considered core abstractions. In general, these early decisions can lead to brittle systems by forcing early decisions which become integral to the system and force abstractions into concrete implementations. In addition, mediator patterns also require that the me-

diator understand how different processing nodes interact to effectively control and coordinate. Finally, the mediator pattern is less generalized than typical service oriented architecture patterns which may include mediators as a part of the service infrastructure.

3.1.1.9 Remote Operations Control Interface (ROCI)

ROCI [Chaimowicz et al., 2003] is a programming framework developed by the University of Pennsylvania to support integration of distributed sensors and actuators in support of multiple autonomous air and ground robots. It supports "injectable" modules, i.e., modules that may be loaded via configuration during runtime. The configuration system relies on XML to describe the configuration settings. ROCI makes use of modules which represent the processing of a form of input data to derive another form of output data as a black box. The module makes no assumptions about the source of the input or the consumer of the output. Modules are further organized into tasks which represent collection instances of modules and the interactions between those instances. ROCI uses a communication connection called "pins" which represent communication endpoints on the modules. This model is similar to a hardware connection model.

ROCI has features that are enabling capabilities for service oriented architectures. Specifically, the specification of module configuration and the dynamic loading ability provided through "injectability" is related to the definition of service descriptors and run-time access to service providers. Additionally, the notion of "pins" is incorporated within service descriptors when associating a service's logical interface with a specific communication channel with well-defined endpoints. We will return to these concepts in Chapter 5 when we define the constitution of service oriented architectures.

3.1.1.10 Controlling Robots with CORBA (CoRoBa)

CoRoBa [Colon et al., 2006] is a development framework intended to allow for integration of distributed robotic control, sensor, and processing components. It is

fundamentally about what its name says: using CORBA. CORBA (Common Object Request Broker Architecture) is an architecture for tying distributed object-oriented components together. For more information refer to [Object Management Group, 2007].

Use of CORBA as a communications method between distributed objects has limitations. First, CORBA only works efficiently for components implemented in object-oriented languages. This limitation may preclude or make difficult the use of legacy components not implemented in an object-oriented paradigm. Second, CORBA relies on being able to define an interface via the Interface Definition Language (IDL) and then compiling that interface into a set of proxy objects for use on both the client and server systems. Compiling an IDL description may be difficult to do if a chosen language does not have a corresponding IDL compiler. Third, CORBA is inherently synchronous since it relies on a request/response sequence for each client request. Mechanisms exist within CORBA to give the appearance of asynchronous behavior, but those mechanisms have been traditionally disregarded due to complications that arise from their use. Fourth, most CORBA implementations are quite large and tailoring the implementation to fit within the constraints of an embedded system is difficult or impossible in some cases. Finally, there are few open source CORBA implementations that are readily supported and commercial CORBA implementations can be very expensive.[5]

3.1.1.11 Mobility Integration Architecture

Mobility is the name given to a software architecture used by iRobot in support of their research robots [iRobot, 2002].[6] The architecture is built around the CORBA object model. Mobility is a layered software architecture using a standard CORBA approach to building distributed object systems and offers the same advantages and disadvantages of both layered and CORBA systems.

[5] One commercial CORBA implementation that the author is intimately familiar with is licensed on a per cpu basis at a rate of several tens of thousands of dollars per cpu, renewable yearly. For low volume production systems, this cost is typically passed directly on to the user or not supportable.

[6] Mobility and the research robots program at iRobot has been discontinued since 2004

Mobility offered one of the first commercial views on developing software for integrating systems on robotic platforms by stating explicitly that the product was intended to address the issues surrounding system integration. Additionally, the approach used in Mobility starts in the right direction of dealing with distributed objects but stops short of offering signicant loose coupling due to the way CORBA is used and the limited scope of the iRobot research robots.

3.1.1.12 Operational Software Components for Advanced Robotics (OSCAR)

OSCAR [Kapoor, 1996] is a programming framework from the University of Texas designed to work specifically with control programs for robotic manipulators. It is built around a componentized layered architecture with a strong focus on kinematic processing necessary for manipulator control. OSCAR is not intended as a comprehensive framework or architecture but provides a useful set of components that can be used in a larger architecture for dealing with manipulators.

3.1.1.13 FAST-Robots

The Framework Architecture for Self-controlled and Teleoperated Robots or FAST-Robots [Kenn et al., 2003] is a framework for providing network control capabilities in support of experimentation with teleoperation and various levels of autonomy. The framework is implemented in C++ and provides a base set of classes from which the user derives a set of specialized objects. These objects are registered within the framework and when execution begins, the framework is in control but with user-defined functionality embedded in the framework.

FAST-Robots is designed specifically to be used as a rapid-prototyping tool for experimentation and is focused on RoboCup Rescue Robots. This framework, while not complete enough to use on its own, can be used as a means of communication between components incorporated within a larger architecture such as a service-oriented architecture.

3.1.1.14 Coupled Layer Architecture for Robotic Autonomy

The Coupled Layer Architecture for Robotic Autonomy or CLARAty [Volpe et al., 2000] is a layered architecture implementing a modification of the typical robotic 3T architecture. This architecture is a product of the Jet Propulsion Laboratory and represents several years of research and development in robotic autonomy. The typical 3T architecture involves a functional, executive, and planner layer with each successive layer adding increased "intelligence." In the CLARAty model, the planner and executive layers are merged into a single layer with the addition of a common database shared between the planner and executive layers.

Since CLARAty is a 3T architecture it suffers from the issues previously described for 3T architectures. However, as with many of the examples in this section, we believe that CLARAty could be usefully included in a much larger architectural description.

3.1.1.15 Robot Operating System (ROS)

ROS [Quiqley et al., 2009] is an open source effort based on work done at Stanford University and Willow Garage[7] with the goal of providing an architectural framework supporting modular, tool-based development for robotic software. The ROS framework provides tools and components supporting interaction patterns between software processes and elements such as messages, topics (broadcast messages), and services (request/reply). ROS is a very loosely coupled architecture making use of service orientation but also permitting other architectures to coexist. In addition, ROS provides a network of repositories that federate other organization's open-source software.

ROS is organized around a package concept. Packages represent the lowest level of decomposition under ROS and can consist of any independent software construct with a well-defined boundary. Examples include configuration files, executables, libraries, or third party software. Packages are further organized into stacks

[7] A company founded in 2006 with the purpose of developing personal robotics and open source robotics software.

which are collections of packages that provide some level of functionality such as navigation. Stacks provide the ROS mechanism for code sharing either through linkage like a library or through runtime access via messaging.

Since ROS is intended to be an operating system for robots, the ROS developers have provided an extensive toolset for navigating, managing, and manipulating packages and stacks. These tools include package and stack creation tools, extensions to Unix shells for navigating the ROS stacks and packages, and tools for creating messages and services for handling these messages. Within ROS, the notion of a service is defined as a pair of request/reply messages. Services are offered by nodes which are computational processes. A robot might contain several nodes representing different types of computation found on a robot such as navigation, path planning, motor control, or a specific type of sensor.

ROS is similar in approach to the service oriented approach described in this thesis, but was released after the initial proposal of this thesis. Additionally, the approach taken in this thesis is intended to be more general than that used in ROS. This difference will be discussed in subsequent chapters, but is related to the way in which users of the architecture elements access those elements and define requirements for use of services. However, we believe ROS and the architecture defined in this thesis represent a convergence of thought within the robotics community that a loose coupling approach to building robotic systems is desirable and increasingly necessary.

3.1.2 SUMMARY OF EXISTING ROBOTIC ARCHITECTURES

As can be seen from the previous paragraphs, there are a considerable number of architectures/frameworks aimed at providing guidance on building software for autonomous systems. Although there are many more systems described in the literature, the ones discussed represent the mainstream of development and are indicative of the approaches taken by researchers in getting to more flexible software. Table 3.1 summarizes the traits of each of the aforementioned systems with respect to the behavioral models and architectural patterns used.

While each of the systems listed in Table 3.1 have their strengths and weak-

Arch. Style / Robot Arch.	Layered	Messaging	Pipeline
Reactive	Subsumption		
Hybrid	4D/RCS 3T CLARAty MARIE ROCI OSCAR FAST-Robots	JAUS JTA CoRoBa Mobility	
Deliberative	Shakey		
Framework	ROS Player Pyro		

Table 3.1: Relationship between robotic behavior models and software architectural styles for existing robotic architectures

nesses, no one of the systems, with the exception of ROS, is sufficiently flexible to permit integration of both new development and legacy components without significant development effort. We believe that a different approach is required to assist all stakeholders in robotic systems development and the research proposed here constitutes the first examination of robotic architectures in the context of a software engineering approach and with respect to a set of quality attributes expected of robotic systems. As of the date of this thesis, other researchers are beginning to explore similar approaches to reducing the difficulty and cost (in both time and money) of building integrated robotic software. We believe that the current practice in commercial software engineering is adequate to address this problem and needs to be shown to be adaptable to the robotics community.

In the next chapter we consider how to compare different architectures or architectural approaches in the context of integration. This comparison will be made through the use of appropriately defined metrics for quantifying the effects of integration on an existing system.

4. SOFTWARE METRICS BACKGROUND

We have stated as a hypothesis that an architectural approach based on service orientation provides a more efficient means of supporting integration on robotic platforms. A natural question to ask now is, "How much better, if any, is a SOA approach over other architectures already in use?" To answer this question in a meaningful way requires a definition of what we mean by "better" in this context. In this chapter, we examine an approach using metrics to answer the question. The chapter starts with a discussion of what "better" means. We then examine existing metrics that might be used and limitations to their use. The chapter ends with the specification of a set of metrics for use in comparing architectural approaches to integration. These metrics are used in Chapter 13 to examine the performance of the SOA for robotics.

4.1 Measurement Context

Metrics are functions mapping measurements into values representing some characteristic of interest [Kaner and Bond, 2004]. This definition is the basis for the Representational Theory of Measurement as used for software metrics [Fenton, 1994]. We will adopt the viewpoint that a metric for an attribute is a function of measured quantities that preserves empirical relations between those quantities. Such a metric is called a *valid representation*. In addition, many transformations between different representations for a metric may exist but we are only interested in those transformations (called admissible transformations) giving valid representations when acting on valid representations [Fenton, 1994]. Based on these properties we understand that statements about measurements are only meaningful when the admissible transformations are truth-preserving when applied to the measures in the statements [Fenton, 1994]. In other words, if we are making statements about length, then those

statements are valid for English system representations and for metric system representations obtained through conversion from English system units.

With these definitions, we can now specify what we are interested in measuring. A service-oriented approach to integration was chosen with the assumption that this approach can reduce the amount of unfamiliar work in unfamiliar code to be done by roboticists. Software integration work requires an understanding of

1. The structure, organization, and functioning of the new code,
2. The usage model of the new code,
3. The structure, organization, and functioning of the existing code[1],
4. The usage model of the existing code,
5. The desired functioning of the integration result,
6. The model for test planning, creation, and execution, and
7. Techniques for identifying, locating, and fixing problems with the integrated code.

These different concepts all contribute to the cognitive complexity of performing integration. Increased code complexity is associated with increased difficulty in working with the code and greater numbers of defects in that code. This increased difficulty results in more time, people, and money to do integration, the potential need for specialized skills, additional maintenance costs, or some combination of these. We are thus looking for metrics that characterize this complexity, allow for pre and post integration analysis, and can give some notion of how the complexity translates into resource, schedule, and financial costs. Such metrics should also provide an ability to make statements about architectures such as

> "Architecture A is preferred over Architecture B because A is less complex than B."

[1] While it may sound strange to state this, in many cases, integration may be done by people who are not completely familiar with the existing code. However, understanding of the existing code is assumed when planning integration activities.

or

> "Architecture A is preferable to Architecture B because using A will take less time, people, and money to add new features than B since A is easier to understand."

We capture the intent of the previous statements in the following definitions for "Architecture Goodness" and "Architecture Fitness."

4.1.1 Definition (Architecture Goodness) *A set of desirable architectural qualities represented by measurable architectural properties*

These qualities are sometimes referred to as *-ilities* or quality attributes [Fabrycky and Blanchard, 1998]. Examples of *-ilities* include testability, maintainability, flexibility, modifiability, and reliability. Defining architectural goodness sets the expectations for an architecture. However, knowing when those expectations have been met requires translating the expectations into a measure of how well a given architecture matches those expectations. For this alignment of expectations with reality we use the term "architectural fitness" and define it as follows:

4.1.2 Definition (Architecture Fitness) *Evaluation of an architecture against the measured values of properties representing the desirable qualities defined for architectural goodness*

Since integration is inherently about the modification of existing software, or modifiability, we expect our metrics to measure changes in complexity through software modifications. In fact, our hypothesis is that service-orientation reduces the overall system complexity encountered during integration by moving difficult integration tasks into standardized components used within a common infrastructure. These standardized components can be created through automatic code generation from simple declarations about what capability the component provides and how that capability is to be used. While the overall system complexity may not necessarily be reduced, the complexity directly encountered by the person doing the integration is significantly reduced.

4.2 Existing Metrics

Service-oriented approaches are distributed specializations of component-based software systems. Thus, as a starting point, we will examine existing metrics for object-oriented software, component-based software, and service-oriented software. However, many of these metrics rely on other metrics representing the length of a function as defined on procedural code such as counts of source code lines (SLOC) or counts of the number of independent execution paths through the source code. The latter is obtained from a consideration of an acyclic, directed graph representing the control flow through the source code and is known as the McCabe Cyclomatic Complexity and denoted by v_g [McCabe, 1976]. To understand these "foundational" metrics we will examine how metrics are categorized and the characteristics most commonly associated with various metrics.

4.2.1 METRICS TAXONOMY

Many different metrics exist to serve a variety of purposes. However, based on the taxonomy of metrics described in Mills [1988] and in Fenton and Pfleeger [1998], all software metrics can be grouped into at least one of three categories. Those categories are

1. Process—metrics associated with models of software development
2. Product—metrics associated with the artifacts of software development
3. Resource—metrics associated with the project or program structure used to manage software development

We will not be concerned with resource and process metrics other than to point out that the results of measurements on development artifacts for specific projects and within specific organizations are dependent upon the effective use of both processes and resources. The process and resource metrics are designed to measure the effectiveness of processes and resources in creating products, while the product metrics are

designed to measure the overall result of those processess and resources as exhibited by the product properties.

Product metrics can be further categorized based on the aspect of the artifact that the metric characterizes, as follows:

1. Size—attempts to capture a notion of the quantity of software. Examples include Source Lines of Code (SLOC) count, function points, object points, and Halstead's code volume.

2. Complexity—attempts to capture a notion of how difficult the software is to understand or manipulate. Examples include McCabe complexity and Henry and Kafura information flow complexity.

3. Quality—attempts to capture a notion of how good or bad the software performs when used. Examples include defect count, mean time between failure (MTBF), and defect repair rate.

Complexity metrics are typically subdivided into computational complexity and psychological or cognitive complexity. Computational complexity is associated with the difficulty of performing specific computations, usually in terms of time and space requirements. Cognitive complexity, however, is the complexity associated with the difficulty in understanding and using software. This difficulty has been associated with the quality of developer performance, quality of the software artifact, and the existence of errors in the artifacts [Zuse, 1990]. In Wang [2009], cognitive complexity is associated with time and workload effects on human cognition (related to software development artifacts) and can be divided into categories based on the sources of complexity as follows:

1. Symbolic—the quantity of symbols associated with an artifact, equivalent to size metrics. Examples include the number of implementation language keywords, datatypes, variable names, or number of characters used.

2. Structural—the quantity of unique symbol arrangements and relationships between those symbol arrangements. An example is the structure of source code

in a particular implementation language for which the symbol arrangements are constrained by the language syntax.

3. Data—the quantity of individual data elements and their movement within an artifact or between artifacts, e.g., class attributes and their use within class methods within an object oriented language.

4. Functional—the quantity of unique arrangements and relationships between structural and data elements within an artifact, e.g., the methods associated with a particular class in an object oriented language.

5. System—the quantity of unique arrangements and relationships between functional elements within a system. Examples include the classes used and the associations between those classes in an object oriented program, the classes and relationships between them in a software component, or the arrangement of systems within a system of systems.

In addition, considerations of time for each of the above categories leads to a grouping of metrics into static or dynamic metrics. Static metrics are those that measure properties of software that exist independent of execution of the software and dynamic metrics measure those properties that exist only upon system execution.

In the remaining sections of this chapter we will look at existing metrics grouped by programming model and define a set of metrics for comparing architectures. We consider metrics that are a combination of symbolic, structural, and functional complexity with consideration for both static and dynamic characteristics.

4.2.2 OBJECT-ORIENTED SOFTWARE METRICS

The most familiar object-oriented software metrics are the suite of metrics defined by Chidamber and Kemerer [1994] and referred to as "CK Metrics." Much of the subsequent work on object-oriented software metrics derives from the CK metrics, e.g., Perepletchikov et al. [2007b], Vernazza et al. [2000], Lilienthal [2009]. The CK metrics are defined as follows:

1. Weighted Methods per Class (WMC)—the sum of the complexities of the methods within a class. Frequently, this is computed as just a count of the methods within the class. High values for WMC imply increased complexity because entities with a large number of properties are assumed to be more complex than those with fewer properties.

2. Depth of Inheritance Tree (DIT)—the count of the levels of inheritance from the root to the class being measured. Large values of DIT indicate a deeply nested inheritance tree which is structurally complex. Such deep nesting can be a source for cascade effect change when modifications occur higher up the tree. Additionally, changes in the measured class must take into account all of the ancestor methods and attributes to avoid conflict.

3. Number of Children (NOC)—the count of the immediate subclasses derived from the class being measured. Like DIT, large values of NOC indicate high complexity. Change made in the parent class will ripple down to the children and the potential for conflict is increased when a larger number of derived classes from a single parent are present.

4. Coupling Between Objects (CBO)—the count of the number of other classes to which the measured class has an association other than inheritance. A high degree of coupling between classes increases complexity because changes in a single class have the potential to influence all of the classes coupled to the changed class.

5. Response for a Class (RFC)—for each method in the measured class count the number of methods called external to the class. RFC is the sum of the counts for all methods within the measured class. A large response set for a given class is another form of coupling complexity with coupling between the methods invoked as a result of a single invocation. This complexity can lead to defects when modifications are made because of missed calls, changed parameters, or other modifications that may invalidate the use of a particular method.

6. Lack of Cohesion in Methods (LCOM)—compare the methods in a class pairwise, counting the number of pairs which use no instance variables in common and separately counting the number of pairs using instance variables in common. The difference between the first count and the second is the LCOM and is set to zero if negative. High values of LCOM indicate that a class may be doing too many different functions. This leads to higher complexity if some of the functionality is not confined to the high LCOM class because making modifications requires finding all occurrences of the functionality. This can lead to increased defects if all occurrences are not accounted for. Additionally, separating out functionality for reuse is not possible for functionality contained in classes with high LCOM values.

The CK metrics are not without problems. Some of these are indicated in Riguzzi [1996]. Much of the subsequent work based on CK metrics is related to correcting some of these problems. Notable in this is the work of Briand et al. [1998, 1999] which attempts to refine measurements and interpretations of coupling and cohesion.

Other metrics approaches for object-oriented software are based on measurements of information flow through the source code. This type of metric was popularized by the work of Henry and Kafura [1981]. Of particular interest is the definition of the complexity of a procedure as $l \times (f_i \times f_o)^2$, where l is the length of the procedure and f_i and f_o are the fan-in and fan-out for the procedure[2], respectively. This metric for complexity combines a metric for the internal complexity, l, and a metric for the external complexity of the procedure based on how strongly coupled the procedure is to its external environment. This inference from coupling to complexity, similar to the CK metric CBO, is based on the idea that the more strongly coupled a procedure is to its environment, the more difficult it will be to comprehend the full effect of a modification to either the environment or the procedure itself. Modifications are

[2]The concepts of fan-in and fan-out are defined by Henry and Kafura in terms of information flows and data structures. Fan-in is the number of information flows into a procedure plus the number of external data structures read by the procedure while fan-out is the number of information flows from the procedure plus the number of external data structures updated by the procedure.

also likely to cause change to ripple through the couplings and such a procedure is considered to be more complex than one with fewer connections.

4.2.3 COMPONENT-BASED SOFTWARE METRICS

Component-based software is a means of constructing large, complex software systems through composition. A component is defined as a self-contained, prefabricated unit of composition capable of being deployed independently by third parties [Kharb and Singh, 2008]. The functionality offered by a component is accessible via well-defined interfaces where the actual implementation of those interfaces is inaccessible to users of the component. Measuring complexity in component-based systems must account for these characteristics.

Complexity metrics for component-based systems vary widely in definition. The metrics defined in Kharb and Singh [2008] treat components as black boxes and focus on the interactions between components by defining metrics for the ratio between incoming and outgoing component interactions, interaction density, total number of interactions, actual number of interactions, and normalized complexity of incoming and outgoing interactions for the complete system. Another approach proposed in Vernazza et al. [2000] defines metrics similar to the CK metrics but with a component interpretation.

In Jiao et al. [2008], complexity metrics are defined for components, composites, and dependencies. Complexity for components is defined through the number of operations and the number of parameters defined by a component while dependency complexity is defined through the summed complexities of individual data types defined by a component and based on the position of a type in a data type graph. Composite complexity is defined via a weighted dependency graph where edges between components are dependencies and the edge weights are measures of influence between the components.

The approach outlined in Gill and Balkishan [2008] define a Component Dependency Metric counting the number of dependency paths between a component and all other components in the system. The metric is computed by the ratio of the actual

count of dependency paths to the maximum possible number of paths. The second metric is the Component Interaction Density which is the ratio of the total number of direct interactions between components to the total number of components.

4.2.4 SERVICE-ORIENTED SOFTWARE METRICS

Services as we have defined them are stateless, logical entities accessed via messaging. Metrics for services must be defined to address the characteristics of services as given in the definition. With service-oriented concepts being relatively recently developed, metrics for services are less well-defined.

In Perepletchikov et al. [2007a], the authors propose a formal model for service-oriented computing and then use that model to define a collection of metrics in Perepletchikov et al. [2007b] for measuring the maintainability of service-oriented designs. These metrics are defined as follows:

1. Weighted Intra-Service Coupling Between Elements—count of the inbound and outbound connections between pairs of implementation elements[3] within a service. The weighting is based on the types of the elements in the connection, e.g. procedure, class, interface.

2. Weighted Extra-Service Incoming Coupling of an Element—count of the inbound connections from implementation elements of other services to a single element. The weighting is based on the type of the relationship between the elements

3. Weighted Extra-Service Outgoing Coupling of an Element—count of the outbound connections to implementation elements of other services from a single element

4. Extra-Service Incoming Coupling of Service Interface—count of the number of external elements using a particular service interface

[3] An implementation element is any software construct, other than a service, that would provide the implementation of a service

5. Element to Extra-Service Interface Outgoing Coupling—count of the number of service interface implementations a given element is a part of

6. Service Interface to Intra-Element Coupling—count of the number of direct relationships a service interface has with its implementing elements

7. System Partitioning Factor—the ratio of the count of elements belonging to at least one service to the total number of elements

8. System Purity Factor—the ratio of the count of pairwise intersections of all services with respect to implementation elements resulting in the empty set and the count of all services

9. Response for Operation—count of the implementation elements and other service interfaces executed in response to invocation of an operation in a service interface for all possible parameters

Additionally, eight metrics for computing the previously described metrics for aggregates of services are defined. The metrics defined here are proposed and no actual empirical data is provided for how these have been used in an operational service-oriented system. However, the authors describe validation of the Weighted Intra-Service Coupling Between Elements metric using a property-based software engineering measurement framework but the validation is simply to show that the definition of coupling has not been violated by the metric. These metrics are defined at a very low level of granularity and make assumptions about service implementations that may not always be satisfied, namely that the implementation of a service may have elements that are in use by other service implementations or are called directly by other implementations. Based on this observation, we note that the System Partitioning Factor and the System Purity Factor show some utility as a measure of how cleanly the service interfaces are divided. We also emphasize that the authors have made parallels with some of their metrics and the CK metrics, i.e., the coupling metrics and the Response for Operation metric described above.

Another set of service-oriented complexity metrics are defined in Rud et al. [2006] as

1. Network Cohesion in the System (NCY)—count of the connections between nodes where a connection is a service consumer invoking against a service provider. This metric attempts to capture complexity through the degree of coupling in the system so that large values of the metric correspond to greater complexity.

2. Number of Services Involved in a Compound Service (NSIC)—count of the number of services used to create a compound or composite service (in our terminology). Composite services create new functionality by stringing together existing services. This metric attempts to capture complexity through the size of the composite service as measured by the count of constituent services. Large values for the metric are assumed to correlate to greater complexity.

3. Services Interdependence in the System (SIY)—count of the pairs of services having a dependence relationship. A high degree of interdependence indicates that the service granularity may be too fine. This interdependence contributes to complexity through the necessity of having to ensure that dependencies are satisfied when modifying the system.

A collection of metrics related to the criticality and reliability of services within a system are also defined but we will not discuss them here. We observe that high values of NSIC can be viewed as either good or bad depending upon the perspective. From a system perspective, high values of NSIC can be viewed as a good thing since the complexity of the system is potentially encapsulated in composed services rather than lower level atomic services. From the individual service perspective, high values of NSIC can indicate that modification of any of the constituent services may have a larger impact than anticipated. Additionally, consideration of the SIY metric in conjunction with the NSIC metric may give insight into how well the composed services manage the interdependence between the constituent services at a system level. At

the individual service level, high values of SIY can indicate that modification of the service may be difficult or have impact on all services with dependence relations on the modified service. The above metrics can be seen to be related (at least in principle) to the CK metrics. As in the proposed metrics of Perepletchikov et al. [2007b], the metrics defined here are not supported by empirical data from an operational system and the authors do not present any validation data.

Finally, Hofmeister and Wirtz [2008] define two sets of metrics associated with service-oriented systems. The first set are considered basic measures to be used in formulating more complex metrics and include

1. Number of Services—count of all services. Higher numbers of services are assumed to indicate higher complexity

2. Number of Service Consumers—count of all services functioning only as consumers. No inference is made on the values obtained.

3. Number of Service Providers—count of all services functioning only as providers. No inference is made on the values obtained.

4. Number of Service Aggregators—count of all services functioning as both a consumer and a provider. The values obtained here can potentially overlap with the counts of service providers and consumers.

5. Coupling of Service—count of other services used by the implementation of a service. High values can indicate potential for change impact if any of the included services change their interface.

6. Inter-Service Coupling—count of the number of connections between two given services. This count includes both the incoming and outgoing service request channels. No inference is made on the values obtained.

The second set of metrics are built up from the basic measures and include

1. Service Coupling Factor—sum of the Coupling of Service metric over all services normalized by the Number of Services metric. This metric intends to capture

the overall coupling within the system. Low values indicate a low degree of coupling and are intended to indicate a more modifiable system.

2. System Service Coupling—sum of the Coupling of Service metric over all services normalized by the maximum number of couplings possible in the system as computed by the product of Number of Service Consumers and Number of Service Providers. The metric is intended to capture the amount of interaction between services that do not occur via mediation, i.e., through a service aggregator. High values indicate a complex system because the coupling between services is direct.

3. Extent of Aggregation—ratio of the number of pure consumers coupled with aggregators to the number of pure consumers coupled to all providers, i.e., pure providers and aggregators. Low values represent low aggregation which is assumed to be associated with a more complex system in a way similar to the System Service Coupling.

4. System's Centralization—the difference between consumer coupling and the coupling due to aggregation divided by the overall consumer coupling. An adjustment is made by deducting the count of pure consumers from the denominator value. This adjustment is intended to compensate for the fact that all service consumers are coupled by definition to service providers. An additional adjustment is made in the numerator to penalize excessive use of aggregation. The metric intends to capture the degree of centralization, as implemented through aggregation, used to manage complexity. High values indicate a higher degree of centralization and the assumption is that under most conditions, this is favorable.

5. Density of Aggregation—the sum over all aggregators of the logarithm of the ratio of the number of potential inbound service calls to the total number of potential inbound and outbound calls. This metric is intended to capture how much aggregation use is devoted to true aggregation since aggregators may be

used for other purposes. The metric must be used in conjunction with the System's Centralization metric to capture the "goodness" of the aggregation with respect to centralization.

6. Aggregator Centralization—computed by subtracting the ratio of non-mediating aggregators[4] to total aggregators from one. The metric is intended to capture the degree to which centralization is handled by non-mediating aggregators. This metric is to be used in conjunction with the System's Centralization and Density of Aggregation metrics to support the interpretation of the "goodness" of centralization.

Each of the metrics in the second set are defined to capture an aspect of complexity associated with modifiability within a service-oriented system. The authors have applied these metrics to an industrial project to guide the design of a business process implementation and refer that implementation to a composite application reference architecture. The stated result is that the metrics were of value by helping to identify poor design and providing a rationale for re-design prior to the actual implementation.

There are several issues with this particular set of metrics that make them difficult to consider for use. First, computing many of the metrics requires information or access to the internals of a service implementation. This information may not be available or extremely difficult to obtain. Second, the point of view of the metrics are from within a single organization comprising the enterprise. In reality, the real value of service orientation is that since many enterprises consist of multiple independent organizations, services that are governed by negotiated contracts (Service Level Agreements) around the use and management of those services that cross ownership boundaries enables more efficient interaction [OASIS, 2009] . Third, the interpretation of centralization and aggregation metrics is potentially misleading without consideration of the nature of the enterprise. An enterprise consisting of federated systems, each of which may be considered a separate entity, will not exhibit centralization as expected but will show centralization through the service

[4]A non-mediating aggregator is a service acting as both a consumer and provider but which provides capabilities that go beyond just mediating calls to other services.

mechanisms used to federate the enterprises. Additionally, systems with redundant mechanisms for dealing with reliability and availability will skew metrics for aggregation. Consider the case of a load balancing mechanism. Such a mechanism is an aggregator, can be considered to be a service, and may exist in multiples to provide protection against failure. However, the use of multiples will give misleading values for the aggregation metrics since each load balancer instance is connected to the same set of services.

All of the metrics for service-oriented architectures described above take a common approach to characterizing the complexity of the system through some combination of external service features and a knowledge of the internals of the service implementation. Where those internals are not accessible, these metrics become difficult or impossible to compute. Additionally, the authors in all three sources use the CK metrics directly or as interpreted for components to construct their metrics. Of particular note is that while there is a similarity between components and services, the fundamental difference not captured in the above metrics is that a component's interface is not necessarily a logical interface and by definition a service interface is always a logical interface[5]. In Chapter 6, we define metrics to capture the complexity of using a service-oriented approach and relate that complexity to a cost or level of effort required to make the necessary changes in complexity associated with integration activities.

4.3 Summary of Metric Approaches

The previous sections have identified many different approaches to measuring and computing metrics on software in different forms and on different architectural arrangements of software. A notably missing feature of all of the previous metrics is a connection between an economic measure of the impact of changing the software and the complexity of the software exposed to those making the change. Since integration

[5]A logical interface is one representing functionality that may not directly map into an underlying implementation but which can be made physical through association with location and connection information [].

is inherently about creating a product that delivers value that is higher than the cost of the effort of creating the product, we are seeking an architectural metric that captures an architecture's intrinsic cost per unit of increased capability. In the next chapters we show how such a metric can be computed and in subsequent chapters use actual integrations to show the utility of the metric.

Part II

Methodology

5. SERVICE ORIENTED INTEGRATION APPROACH

In this chapter we describe the design and implementation of our integration architecture based on service-oriented principles. The first section covers the architecture design including the basis for our requirements on the architecture and details of the design. The second section discusses the implementation approach for the architecture and the realization of that approach.

5.1 Architecture Design

This section describes the design of an architecture based on service-oriented principles. We describe the architecture according to accepted guidelines for software architecture documentation and include an analysis of business and technical requirements, the design approach, and the design details.

5.1.1 REQUIREMENTS

Deriving requirements for a general integration architecture is difficult because we are not designing for a specific product or for a particular customer. Instead, we are creating a generalized prototype architecture to illustrate how service orientation can be used to integrate new software capabilities into the system of the robot. Because of this lack of specificity, we do not have a set of clearly delineated stakeholders from which we can derive the requirements on the architecture. However, we can identify general categories of stakeholders and use an understanding of the motivations of typical members of those categories to frame a set of architecture quality goals or *quality attributes*, i.e., desirable characteristics exhibited by the architecture, which constitute architectural goodness for integration purposes. At the intersection of the

quality attributes for all of the stakeholder categories is a core set of quality attributes we will consider fundamental with respect to integration on robotic platforms.

For robotic systems, one categorization of stakeholders is

1. Researcher/Developer
2. Commercial Developer
3. System Integrator
4. End User

In the next few paragraphs, we present a profile of each stakeholder category and identify quality attributes that support the profile description's expectations of the architecture. A description of the quality attributes used below can be found in Appendix B.

5.1.1.1 Researcher/Developer

A researcher in the robotics field seeks to try out various capabilities in a controlled environment with the idea of advancing knowledge and utility in the field of robotics. These experimental variations may be new sensor systems, new programming models, new development languages, or even new mobile platforms, but are generally things for which an appropriate interface or usage model may not exist. Part of the activity of research is to define and justify such interfaces and usage models. Furthermore, researchers work within severe budgetary constraints and so look for solutions that are cost effective and perform well but are less concerned with the ease of implementation. These constraints also drive a need for cross-platform usability to accommodate legacy platforms of varying computing capabilities. A key feature of the majority of work in robotics research is that integration is not the primary focus of the research and so flexibility with respect to faster and easier integration offers a benefit in reduced time and budget spent on integration allowing for more time and budget available for the central research concern.

From this profile for the Researcher/Developer we can identify the following quality attributes:

1. Scalability
2. Modifiability
3. Extensibility
4. Performance

5.1.1.2 Commercial Developer

A commercial developer is in the business of making money and so is concerned with time-to-market and ease of adding new features as needed to compete in the marketplace. Additionally, a commercial developer looks for areas of differentiation to make their product more appealing than their competition. Commercial developers are also very concerned about the overall lifecycle cost of the platform. If maintenance costs are extremely high, the product may present a loss for the corporation and thus not be a product that stays long in the marketplace. Further complications arise for the commercial developer in the market impression of the product. A product that is viewed as being buggy, hard to use, or difficult and expensive to maintain will not be widely accepted and will again represent a loss for the commercial developer. Like the researcher, the commercial developer is also interested in the ability to reuse the same software components on platforms of varying computing capabilities while achieving comparable performance levels.

From this profile for the Commercial Developer we can identify the following quality attributes:

1. Scalability
2. Maintainability
3. Reliability
4. Extensibility

5. Modifiability

6. Performance

5.1.1.3 System Integrator

A system integrator (SI) is responsible for bringing various previously developed components together into a coherent functioning system. In general, system integrators do not develop new components but prefer to contract out any such development when required while keeping cost low so as to maximize the margin on generally low margin fixed price contracts. This ability to outsource is facilitated when the SI has components that can be handed to the third party development group intact or with a simplified interface. Further enhancements to the SI ability are through the ability to provide components without a strong dependence on the computing platform. One area where system integrators differ from commercial developers is that frequently a system integrator must use a robotic platform in the context of a larger system. This larger system context places additional demands on the underlying computational resources and requires the SI to worry about the impact of each integrated component on the overall performance of the system.

From this profile for the System Integrator we can identify the following quality attributes:

1. Scalability

2. Maintainability

3. Reliability

4. Extensibility

5. Modifiability

6. Performance

7. Ease of use

5.1.1.4 End User

An End User does not generally care about the software inside a robotic platform other than how well it functions and how easily new features can be added when needed. End Users also worry about cost and reliability since an expensive platform that does not function reliably is a waste of money. Reliability is important to End Users since maintenance costs and frequent maintenance activities may make the platform unusable. Maintenance costs also can make platform size a concern for the user. A large platform tends to embody hardware components that cost more to replace while smaller platforms may make use of more readily available, cheaper components. This is not always the case since some components needed in the sizes suitable for smaller platforms may be extremely costly. End Users are also generally concerned with how well the system performs since a reliable system that cannot complete tasks in a reasonable amount of time is worthless.

From this profile for the End User we can identify the following quality attributes:

1. Reliability
2. Maintainability
3. Performance
4. Ease of use

5.1.1.5 Stakeholder Quality Attributes Summary

In the following table, we have summarized the quality attribute expectations for each of the stakeholder categories. From Table 5.1 we observe that the single quality attribute present in all stakeholder categories is performance. However, if we consider that the End User derives benefit from the outcome of the work provided by the other stakeholder categories and does not interact directly with the architecture, then we can identify three additional quality attributes in common across the

	Researcher Developer	Commercial Developer	System Integrator	End User
Extensibility	X	X	X	
Maintainability		X	X	X
Modifiability	X	X	X	
Performance	X	X	X	X
Reliability		X	X	X
Scalability	X	X	X	
Usability			X	X

Table 5.1: Stakeholder quality attributes

Researcher/Developer, Commercial Developer, and System Integrator stakeholders. Specifically, those quality attributes are

1. Extensibility,
2. Modifiability, and
3. Scalability

From these quality attributes and the stakeholder categories we can derive requirements that an architecture supporting integration on robotic systems must provide mechanisms to enable changes to the system functionality through the addition of new components or through modification of the existing components. Additionally,

the architecture must provide mechanisms that allow the system to be scaled for use under differing resource size constraints. This latter requirement does not say that the architecture should support the same functionality on the small scale as on the large scale but that the integration mechanisms should be consistent or equivalent at the different scales.

In the next section, we describe the design of our architecture and show how the stated requirements are met by the particular choice of service orientation in the architecture.

5.1.2 DESIGN

5.1.2.1 Service Oriented Architectures

The design of our architecture is based on the view of a robotic platform as an enterprise system. Enterprise systems consist of a collection of independent, co-operating elements interacting to provide value to the enterprise as a whole rather than the individual element. The specific model of this cooperative interaction we use is called "service orientation" because the elements of the enterprise interact by providing and using "services." Architectures based on service orientation are called Service Oriented Architectures or *SOA*. Service orientation is an example of the "Service" and "Collaboration" architecture patterns. All interactions occurring across the enterprise are considered to be through the offering and consumption of services. These service offerings are accessible through well-defined and published interfaces. The entity publishing such an interface is termed a "service provider" and the element making use of the interface to access a service is termed the "service consumer." An important consideration to note is that SOA is a model for structuring relationships between elements of the enterprise and should not be considered a product that can be dropped into place or purchased but rather as defined in OASIS [2009, p.8, chapter 2]

> "Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."

From a software perspective a service is a piece of computing functionality accessible through the interface published by the service provider. The functionality may be a low level function such as a measure conversion or it may be a higher level function such as processing a loan application. A service may be made up other services accessed in a particular sequence or based on outcomes of earlier results or the occurrences of specific events. Such services are termed "composed services" with the first type described as "orchestrated" and the second type as "choreographed." A fundamental characteristic of services is that a potential service consumer must be capable of locating services through a mechanism called "discovery" enabled by a service provider publishing details of the service interface into a "service repository."

Enabling a SOA requires the presence of some key elements including infrastructure to support the interaction mechanisms and identification of candidate services. Additionally, a set of services provided by the infrastructure known as "infrastructure services" are considered essential to provide capabilities such as logging, health and status monitoring, performance monitoring, service repositories supporting service identification, and other general capabilities required by all service participants. One mechanism for supporting service interactions is called an "Enterprise Service Bus" or ESB. An ESB provides the underlying communication capabilities and infrastructure services in support of those capabilities. ESBs may consist of multiple communication protocols with each protocol carried on its own bus or an ESB may consist of multiple ESBs connected through protocol bridging services. Service requests originating in one bus accessing services in other buses are mediated by the protocol bridging services. In this way, ESBs can be expanded to include larger portions of an enterprise or to federate separate enterprises into a larger enterprise.

For robotic platforms, the enterprise perspective is appropriate because it allows for the identification of independent elements in the platform interacting with each other to create functionality that make up the robot capabilities. In the next few paragraphs, we will expand this concept into the design of a service-oriented architecture supporting the quality attributes of extensibility, modifiability, performance, and scalability.

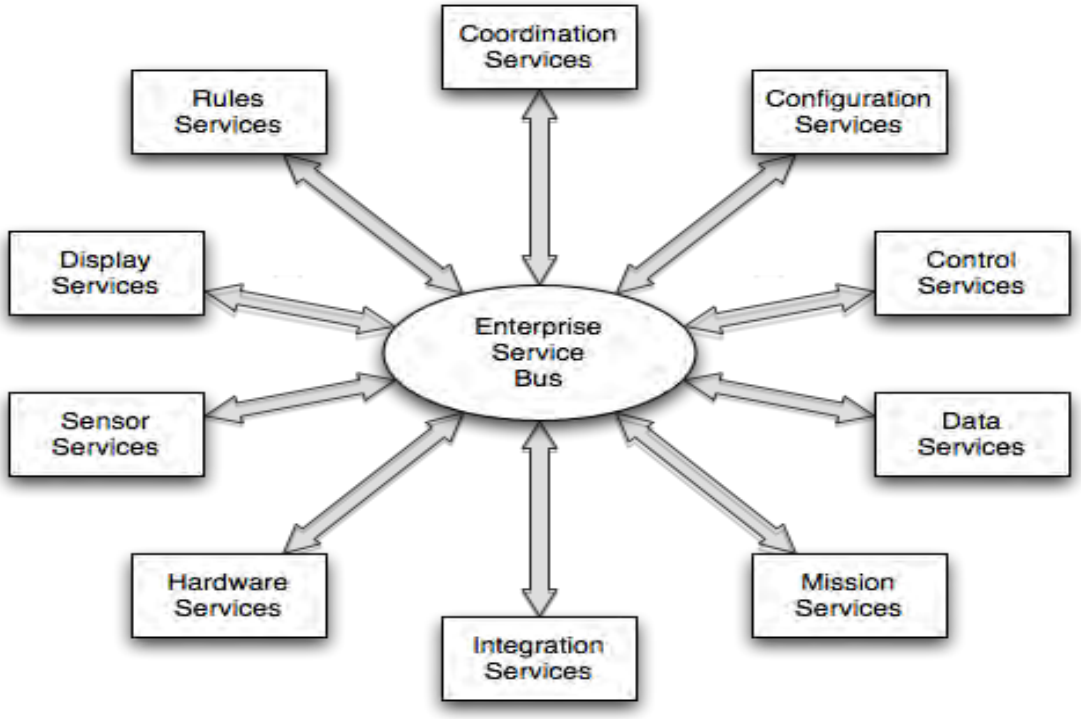

Figure 5.1: Robotic system service categories including the ESB

5.1.2.2 Robotic Services

Candidate services for a robotic enterprise can be identified by observing that robots consist of a disparate collection of sensors, controllers, other hardware, associated software drivers for the hardware, support libraries, and application code. All of these elements work together to give the robot its capabilities. We use the types of elements making up the robot to categorize the types of services available on a robotic platform. This categorization is depicted in Figure 5.1 below and a description of each "service package" follows.

Coordination Services Infrastructure services related to management of the enterprise. This management includes ensuring that mission resources are available, data routing is handled, configuration data is applied, and coordinates the interaction between other services to ensure that the system is functional and the right resources are available at the right time. Besides the duties listed above, the coordination of service composition functions such as orchestration, fault

management, and health status checking of services are handled by infrastructure services..

Configuration Services Services for the use and management of configuration data in the system. These services provide a means to isolate configuration information in such a way that configurable properties of the system and the individual services may be changed in the following ways:

1. Statically, at compile time — used for those basic properties of other services that are immutable across platforms.
2. Dynamically, at system start — used to initialize any configuration information and allow for system customization by changing its configuration at startup.
3. Dynamically, during execution — allows the system to reconfigure without undergoing a restart; requires careful management since some combinations of configuration values may lead to an unstable or inoperable system.

In addition to managing configuration information, this service category can provide services to handle persistence. These persistence services allow for critical system state data to be written out to permanent store when the data changes or when the context the data is useful in has changed. This persistence mechanism is natural since we can envision dynamic configuration information as persisted execution state data and retrieval of the configuration data is the retrieval of persisted state information.

Control Services Services acting as abstractions for control mechanisms used in the platform. This includes steering mechanisms, servo controls, actuator controls, telescoping arm controls, camera controls, sensor controls, and motor controls.

Mission Services Services to define and manage missions which are specific sets of sensors, sensor integrations, hardware, display capabilities, and path definitions

(if needed) combined into a single entity known as the *Mission*. Mission services resemble and can be realized as choreographed and orchestrated services.

Data Services Services providing an abstraction of data streams from sources. The source might be an external sensor such as a camera, an internal sensor such as an encoder disk, or an external data source such as a command stream from a remote controller. A data stream is associated with a particular source to facilitate use of the data but we distinguish between the data provided by the source and the source itself. Some data streams will have a specific destination such as a command sequence to the controller function and such data streams will have a notion of source and sink (destination). Data services facilitate operations such as changing the data sink on the fly or adding additional data streams connected to the same sink.

Integration Services Services to encapsulate data integration or fusion schemes. These services are intended to provide the ability to easily dynamically switch in, switch out, and chain data integration algorithms. For instance, we might want to apply various video processing techniques to a data stream originating from a video source, integrate that modified stream with a data stream from a gas sensor, and finally output a third data stream consisting of the processed video data and the gas data. A consumer for this service might be a display service or a mission service. Additionally, we might want to apply various statistical manipulations to the video data and be able to switch this capability in and out of the processing chain at will.

Hardware Services Services to encapsulate hardware access. These services provide a typical hardware abstraction layer (HAL) as used in most embedded devices but do so through a service interface. The services are responsible for ensuring that all available hardware is configured properly, ensuring that hardware capable of heartbeat monitoring is monitored, and that hardware faults are properly handled before reaching the other software services (if necessary).

Sensor Services Services to encapsulate sensor managment, discovery, and use. This includes sensors for monitoring the internal state of the platform and any sensors used to sense the external environment. Note that we have separated the data provided by the sensors from information about the sensor itself. This separation provides flexibility in how we use both the sensor and the sensor's data. One use of this separation is for a data stream not associated with any sensor, such as a simulated data stream.

Display Services Services to encapsulate display attributes and display management capabilities. These services ensure that the presentation of any data or images conforms to the available display hardware capabilities. Since it is an abstraction of a presentation layer, display services can also be used to configure and manage non-visual presentation of data such as audible data, haptic data or other sensor signals.

Rules Services Services to make use of rule inference engines. These services provide a capability similar to business rules engines used in business software processes. The use of rules services provides the capability to predefine certain behaviours in terms of inference rules. So, for example, we could specify rules about how the Display Service configures itself based on information provided by the Hardware Service without hard-coding this information into the Display Service. Thus, a rules service provides a means of implementing extensibility qualities in the system.

5.1.2.3 Other Architecture Elements

Service consumers need some means of discovering services available for use and service providers need some way to make known what services are offered. These issues are addressed via the "discoverability" mechanisms provided in the infrastructure of the SOA. For our architecture, we initially specify the use of location services as a discoverability mechanism. These location services provide a level of indirection that supports decoupling of the service consumer from the service provider. This de-

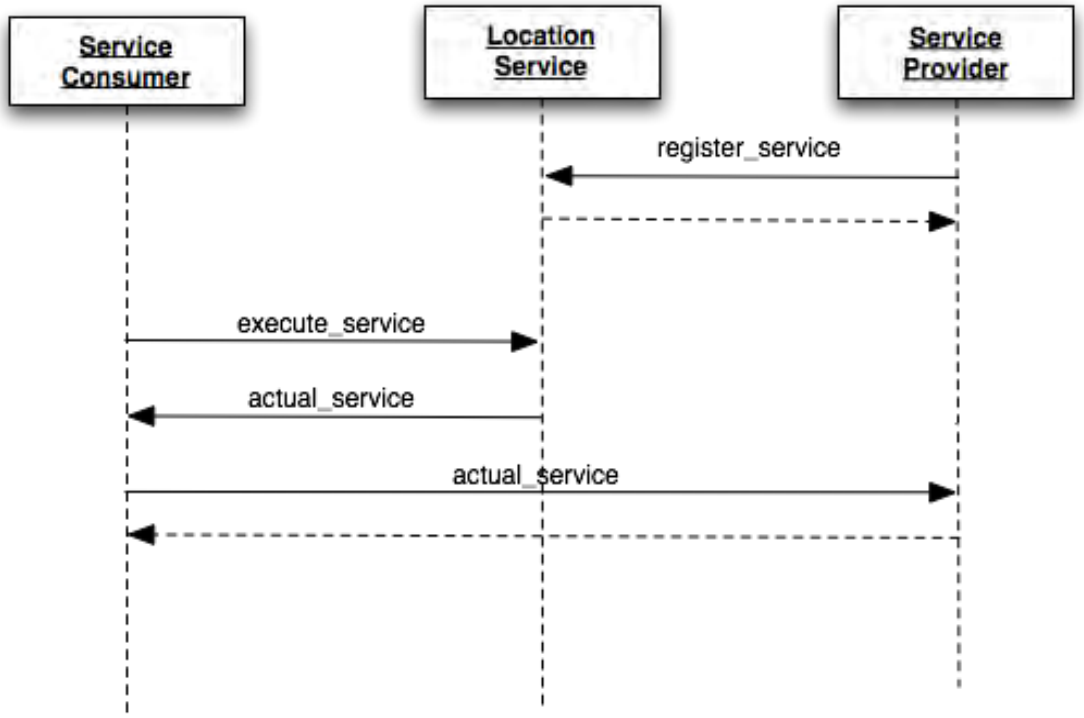

Figure 5.2: Call Sequence for using a location service to access a service

coupling provides flexibility in the choice of discovery mechanism and allows extension of the architecture beyond the immediate enterprise boundaries since most location service implementations support a form of federation. This federation allows different location services to function in collaboration and provide the appearance of a single integrated service. An example of the use of a location service is provided in Figure 5.2. Additionally, since robotic platforms integrate software from various sources and may not have access to the source code, the architecture must provide some means of access to this legacy functionality in ways that do not require modifying existing source code. This problem is largely resolved through the definition of the service interface and packaging of infrastructure and service handling code. Finally, patterns of interaction exist between service participants. Since SOA interactions are effected by messaging between the participants, these interaction patterns are called Message Exchange Patterns (MEPs). Several types of MEPs exist but the most common ones are [Erl, 2009a], [Josuttis, 2007]

1. Request/Response including synchronous and asynchronous variations
2. Publish/Subscribe
3. Oneway or Fire-And-Forget including single destination, multi-cast, and broadcast variations
4. Lightweight Events
5. Point-to-Point

Note that the Lightweight Events and Point-to-Point patterns are variants of Oneway and Request/Response. We can assume that different service participants will have different MEP requirements and so the architecture must provide support for those MEPs in use and an ability to add new MEPs as they are encountered or needed. The use of messaging services such as Java Messaging Service (JMS), CORBA, IBM MQSeries™, and ZeroC IceStorm™ provide capabilities for most of these patterns and also provide capabilities to form more complex MEPs from the basic patterns. With these design concepts and constraints specified, we look at the implementation of the architecture described above in the next section.

5.2 Architecture Implementation

We turn now to the implementation of the design concepts previously described. The implementation consists of several phases including

1. Messaging infrastructure selection
2. Connection protocol abstraction
3. MEP creation
4. Service creation

The creation of services was done in the context of specific real world projects and so service identification was performed based on the usage requirements of the projects. We describe the service implementations for these projects in Parts III and IV.

5.2.1 BASIC INFRASTRUCTURE IMPLEMENTATION

5.2.1.1 Messaging Infrastructure

We began the implementation with a review of candidate messaging systems to provide our basic interaction infrastructure. While it may be tempting to create such an infrastructure from the ground up, the issues and pitfalls associated with building a reliable, fully capable, and supported messaging system make such an approach neither time nor resource effective. This concern is validated if we consider the availability and quality of existing messaging systems. Thus, we evaluated JMS, CORBA, and ZeroC IceStorm™ as candidate messaging systems. These three technologies represent open source and freely available capabilities that are well-supported and well-documented.

Java Messaging System (JMS), as its name suggests, is primarily focused on messaging in Java-based systems. JMS can be used with other systems but the interfaces are more difficult to work with and the addition of translation between Java and non-Java systems adds additional latency to roundtrip request times. Additionally, the ability of JMS implementations to be scaled down for use in constrained computing environments is limited.

CORBA represents a mature technology for working with object oriented systems in an implementation language independent manner. Open source implementations of CORBA are available such as the ACE-TAO implementation. CORBA is implemented according to an Object Management Group (OMG) specification and so enjoys some level of standardization. However, this is also a drawback for CORBA. The OMG is a consortium standards group and so commercial participants must all come to agreement on the content of the specifications before acceptance. This "group think" has a tendency to leave some areas of the specification ambiguous as the commercial participants look to forward their own agendas to protect their investments or market share. Finally, CORBA scalability to constrained computing environments is limited. The ACE-TAO implementation has the ability to be stripped down to a bare bones implementation but even that minimal configuration is still quite large.

The ZeroC Ice™ family of products from ZeroC are relative newcomers to the middleware technology space. However, the company was founded by former CORBA experts and was founded on the premise that while CORBA represents a useful approach to connecting distributed computing capabilities, it suffers from the limitations described above and so they set out to correct those limitations by dropping adherence to the CORBA specification. The ZeroC product line includes the ZeroC Ice™ product which provides the basic distributed, point-to-point capability, the ZeroC IceStorm™ product to provide a messaging capability with message quality of service attributes, and IceBox™ which provides a capability to start other services on demand. The ZeroC IceStorm™ product provides all of the capabilities required to support service orientation using messaging including guaranteed delivery, messaging mechanisms to enable MEP creation, topic creation, federation, and persistent topics and messaging. Additionally, ZeroC has started offering products specifically targeted to providing service oriented architecture infrastructure and products that are designed to be used in constrained computing environments.

Based on the limitations of JMS and CORBA, and the capabilities of the ZeroC product line, we chose the ZeroC IceStorm™ as the basis technology for the architecture infrastructure. All of the implementation details discussed subsequently are built on top of the ZeroC IceStorm™ products. However, that choice can be changed or added to because of the implementation approach for creating connections. Use of ZeroC products requires use of their Interface Description Language called "Slice." All components that require direct interaction with the messaging infrastructure are described in the Slice language and then converted by a compiler into source code for compilation in a target implementation language.

5.2.1.2 Connection Protocol

Despite the messaging infrastructure technology selection above, we wanted to keep the details of that technology hidden from service participants and enable an ability to switch out or add additional technologies. One way of doing this encapsulation is to abstract the connection in such a way that the infrastructure provider/maintainer

creates specific instances of connections but the interface remains the same to users. This is illustrated in Figure 5.3 below. In Figure 5.3, the class relationships hold for

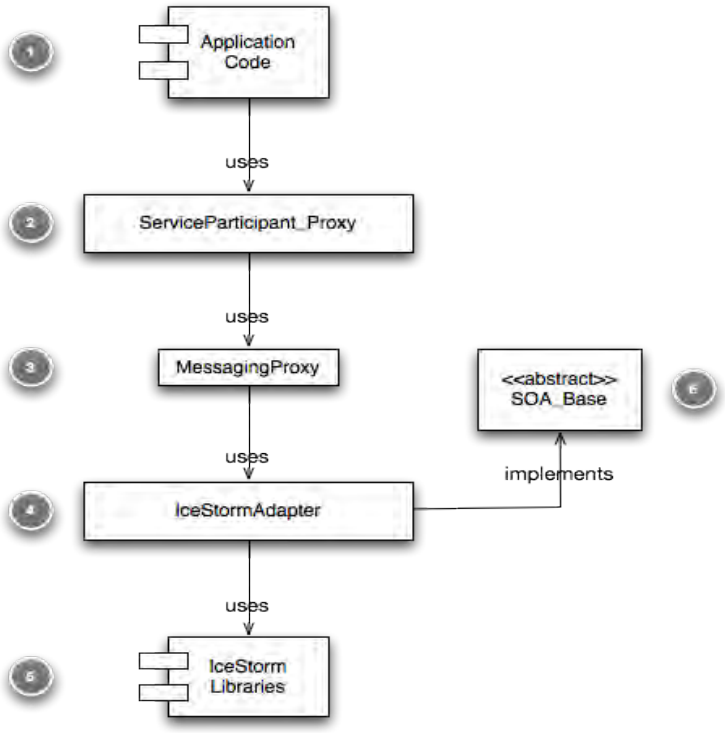

Figure 5.3: Representation of the connection component

both the service consumer and the service provider. The description of the classes is

1. Application Code—Consumer or provider of the service functionality

2. ServiceParticipant_Proxy—Call interceptor to hide the call details for the service functionality. This class creates the appearance of local calls for both consumer and provider.

3. MessagingProxy—Handles access to the messaging infrastructure and implements the appropriate message exchange protocols

4. IceStormAdapter—Connection wrapper implemented over the Ice™ and IceStorm ™ protocol

5. IceStorm Libraries—Implementations provided by ZeroC for the Ice™ protocol and IceStorm™ functionality

6. SOA_Base—Interface definition for the abstract connection

The interface definition for SOA_Base consists of methods that support the use of a transport protocol. The Slice definition of SOA_Base is

```
interface SOA_Base
{
  // Apply whatever configuration or setup is required to prepare
  // the protocol object for connection
  void configure();

  // Make a connection and be ready to start communications
  void connect();

  // Reuse an existing connection that has been disconnected
  void reconnect();

  // Disconnect and stop communications. This call should not
  // remove the resources related to the connection so that
  // connect() or reconnect() can reuse the same connection.
  void disconnect();

  // Shutdown the connection, doing whatever is necessary to
  // clean up any resources still in use. After this call,
  // connect() or reconnect() cannot be called on the connection
  void shutdown();

  // Ping the other end. Intended for status check or for connection
  // check prior to executing a request. This may be a
  // no-op for some protocols.
  void ping();
};
```

The IceStormAdapter class implements the SOA_Base interface using the capabilities provided by the ZeroC Ice™ and ZeroC IceStorm™ libraries. If we needed to add support for JMS, CORBA, or even CANBus, we would create a new protocol adapter implementing SOA_Base using the libraries provided by the specific protocol. This allows us to keep multiple protocols available during runtime to handle different types of service calls. The call sequence with the connection component is shown in Figure 5.4. In the sequence in Figure 5.4, the CallProxy and MessagingProxy change based on the service interface but in the implementation the change involves only the service name, service call parameters, and return type. Because of this consistency we can

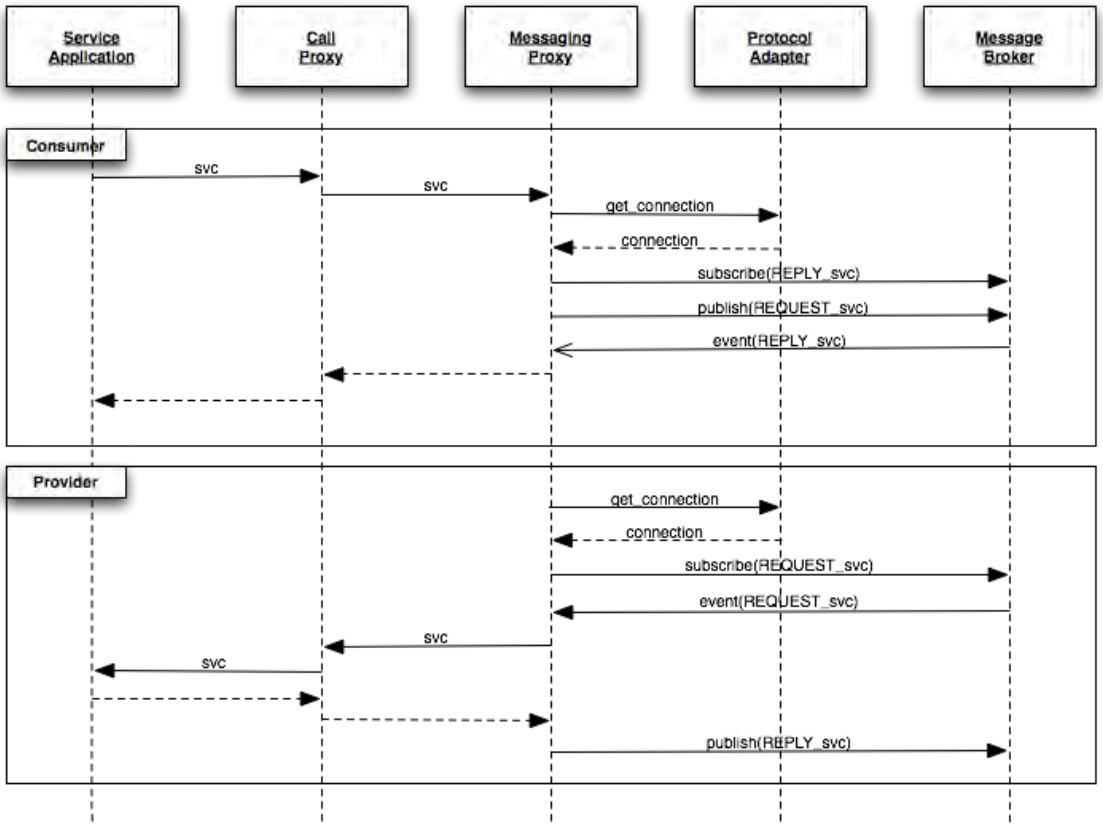

Figure 5.4: Call sequence for consumer and provider

define boiler plate code that can be reused to generate new services. Additionally, the ProtocolAdapter and MessageBroker only change if the protocol is to be changed and in some cases the MessageBroker component may not exist. These differences are accounted for in the ProtocolAdapter and in the MessageProxy so that the addition or removal of specific protocols has only a very limited impact.

5.2.1.3 Service Description

As described in previous paragraphs, much of the code surrounding the actual service request is standard and does not change from service to service. Because of this consistency we built a code generator in Python that takes an XML description of the service called a service profile or service descriptor and generates the CallProxy and MessagingProxy. Since the consumer and the provider have different needs for how these two classes are used, the service profile accounts for each separately. This

also allows different consumer and provider types to make use of the same profile. The XML Schema description of the service profile is given in Appendix A.

The code generator takes the service profile as input and generates the Slice, source, and makefiles necessary to make a service provider or service consumer. The choice of how the service connection is packaged (as a standalone executable or as a shared object file) depends upon how the consumer or provider will access the service functionality. There are several options and the approach to handling each is slightly different. We describe four of the most likely packaging forms here in terms of how the consumer and provider implementations are accessed.

Provider as executable The service connection is created as a standalone executable which calls the implementation executable.

Provider as library The service connection is created as an executable linking in the implementation library.

Consumer, no source The service connection wraps the shared library that would have been used locally and the CallProxy is the wrapper

Consumer, source The service connection is provided as a separate shared library linked into the consumer.

The implementation of the SOA approach provides a basic ESB capability along with a mechanism for creating services, publishing their existence, and discovery by consumers. To compare the SOA implementation with other common integration techniques used in robotics, we constructed integrations using the Player framework [Gerkey et al., 2001] and a basic internet socket approach using point-to-point connections. The next sections describe those implementations as a basis for comparison against the SOA approach.

5.3 Player

Player [Gerkey et al., 2001] provides device abstractions for the devices and many algorithms that are most commonly found on robots. Three primary entities

form the abstractions:

- Interface—a device abstraction through which clients interact with the physical device or the algorithm;
- Driver—the hardware abstraction interacting directly with the physical device or the algorithm;
- Device—the binding of a driver to an interface and identified by a device address.

Clients make calls on interfaces. Those interfaces are implemented by devices which link to drivers. The drivers communicate with the physical device or possibly an algorithm. The client request and subsequent return handling is facilitated by a PlayerClient object which wraps the details of the communication with the server. Interfaces are registered with the Player server which manages client calls and driver execution. Thus, Player is a client-server technology, but many of the communication details are hidden from the user and can be delivered as runtime plugins.

A standard set of interfaces for a large set of device types used in robotics are defined and included in the Player distribution. When adding new drivers, the Player documentation recommends trying to make use of existing interfaces before creating new ones. Within the interfaces included in Player, the "opaque" interface is intended for user-defined messages. However, this interface just ships raw byte data between the client and server, requiring significant coding to map the byte data to structured data. Instead of using the "opaque" interface to do the integration tasks, we have created a new interface and driver plugin for each of the integration tasks and used Player's handling of on-the-wire structured data. This approach is closer to what would be done in a real integration task and so makes for better comparison.

Creating a new interface for use as a plugin in Player requires the following tasks:

1. Create an interface definition using the Player interface definition format.
2. Use the code generator supplied by Player to create source code from the interface definition.

3. Create the interface plugin, i.e., the client proxy.

4. Create the driver plugin.

5. Create or modify the client to use the new interface.

6. Create a CMake task to generate and build the new interface and driver plugins and the modified client.

From our metrics perspective, the interface plugin and the driver plugin belong to the *framework* locations since they are associated with and are only accessible through the Player client proxy or the Player server. Client changes belong to the *base* location. The device driver supplied with the physical device or the algorithm implementation belong to the *new* location. We do not count the interface definition as executable code: the Player interface definition format uses C syntax, but since the code defined in the interface definition is copied into the new plugin header file through code generation, those lines of code would be double counted if we included the interface definition in counting.

In the next section we describe the implementation approach using the internet socket point-to-point (P2P) approach.

5.4 P2P

In point-to-point integration, the client establishes a connection directly to a server without using an indirect addressing scheme. The details of how this connection is made may differ from implementation to implementation, but the end result is that the client and the server are tightly coupled through agreed upon addresses and port numbers. This coupling means that moving the server to a new address may invalidate the client's use of that server. Because P2P integration is widely used, libraries and other toolkits have been created for many of the most commonly encountered implementation languages to facilitate the development and execution tasks associated with using remotely invoked functionality. Examples of these libraries and toolkits include the ZeroC Ice product [Zeroc, 2010], CORBA [Object

Management Group, 2008], Microsoft's DCOM [Microsoft Corporation, 2010], Java RMI [Oracle, 2010], and the Adaptive Communication Environment (ACE) [Douglas Schmidt, 2010]. Note that these examples are fundamentally object oriented and some may not be usable when working with a procedural language.

To make a comparison, we used a basic streaming client and server based on internet sockets as a starting point. This approach is the most basic for connecting a client and server residing on different machines. The utility of this approach is that examples abound and are easy to adapt for use. We have taken a representative example from Hall [2009b] and adapted it for use in the integration tasks. The client and server are written in C and have been put into the public domain for use without license restrictions [Hall, 2009a].

6. COMPARATIVE METRICS APPROACH

We are interested in understanding how various architectures compare when used for integration, so we define a set of metrics here that can be used in making the comparison. Different architectures are defined with different granularities using different terms such as functions, objects, components, or services. Comparisons based on metrics must account for the mismatch between how the metrics are defined and interpreted for these different granularities. Also, different architectures employ different strategies for hiding complexity. To capture this complexity hiding, we want metrics that associate complexity with a location within the architecture. Finally, since our service-oriented architecture has strong dynamic characteristics, our metrics must distinguish between static and dynamic complexity. Additionally, we want a quantification of how well the architecture supports integration. An important point to note is that in any metrics evaluation of an architecture, we will not be measuring the actual architecture, which is an abstract concept, but instead taking measurements on a representation of that architecture. That representation may be in the form of a set of documents describing the architecture or the representation may be in a system implementation based on the architecture. Since evaluation of documentation is difficult at best, we have chosen to base our metrics on a system implementation representing the architecture. This approach assumes that the system implementation is a faithful representation of that architecture but using this approach provides a better basis for making quantitative statements about the architectures under consideration.

6.0.1 PRELIMINARIES

To simplify the discussion, we refer to functions, objects, components, services, or systems as *software elements* or just *elements*. The term *base* refers to the existing

software elements, *new* refers to the new functionality to be integrated, and *framework* refers to elements provided for or used to facilitate the integration. The term *afferent* is associated with flows into an element and the term *efferent* refers to flows out of an element. We define static and dynamic characteristics here to make clear our use. Let t represent time. For any software system, execution of the system has both a start time, t_0 and an end time, T. More precisely, the execution time of the entire system is defined as $t_0 \leq t \leq T$. The dynamic characteristics of the system are those that only exist during the execution period and which may not come into existence until after t_0, go out of existence before T, and possibly occur multiple times in a single execution. Static characteristics are those that exist $\forall t$.

Our metrics are defined on the changes to system elements required to integrate new functionality within a particular architecture. The change set of an element is the set of all additions, removals, and modifications to that element associated with a specific task. For the collections of change sets created during a particular integration activity, we use the following notation:

B	set of all change sets in the base
$b_i \in B$	change set for an element in the base
N	set of all change sets in the new functionality
$n_i \in N$	change set for an element in the new functionality
F	set of all change sets in the framework
$f_i \in F$	change set for an element in the framework

We turn now to defining our metrics and will use the previous definitions as a way of localizing each of the metrics.

6.0.2 COUPLING METRICS

External coupling between elements was previously identified as contributing to difficulty in modifying a system through increasing the structural complexity of the system. We consider, however, that static coupling has a greater impact on the structural complexity than dynamic coupling. Dynamic coupling allows for modification of the overall system without necessarily requiring modification to the internal

structure of the system elements whereas static coupling is created by fixing relationships between elements within the internal structure. For this reason, we will be interested in architectures that exhibit nearly constant static coupling with increases in dynamic coupling following an integration activity. We define such an architecture as "loosely coupled" and, as indicated in previous chapters, loose coupling is a desirable characteristic of architectures supporting integration[1].

Coupling of software elements is an important part of integration since this is the primary way in which in new functionality is introduced. The coupling may occur through the use of API calls, use of data structures, class associations (either *is-a* or *uses* associations) or through messages passed between the software elements. This external structural or relational coupling can be divided into *afferent* coupling and *efferent* coupling. We define afferent coupling as the number of incoming links to a given software element and efferent coupling is the number of outbound links. Note that we do not specify the nature of the link. A link might be through use of a data structure or through a call to a class method. These are similar to Henry and Kafura's use of *fan-in* and *fan-out* [Henry and Kafura, 1981].

Computing the static coupling metrics defined below is straightforward. However, computing the values for the dynamic coupling requires care since the value will fluctuate between a minimum (zero at system start) and maximum value (all possible dynamic connections realized). For a specific application of an architecture, the dynamic coupling will fluctuate about some average value during real operation and this average would be used. However, since we want to make predictive comparisons, running real systems or simulations of systems to collect data is of little help. For our purposes then, we make the convention that the dynamic coupling metrics are computed based on the expected connections made to use the integrated functionality during execution of the system. This means that the values of the dynamic coupling metrics for the pre-integration architecture are zero since the integrated functionality is not present.

[1]This definition is not meant to suggest that dynamic coupling is a sufficient condition for loose coupling but that we are treating it as a necessary and measurable aspect of loose coupling

6.0.2.1 Static Afferent Coupling Change Metrics

We denote the static afferent coupling metric as s_a and parametrize it with the location, $\mathcal{L}$, where $\mathcal{L}$ can take the values B, N, or F. The metric for a single location is defined as

$$\Delta s_a(\mathcal{L}) = s_a(\mathcal{L})_{post} - s_a(\mathcal{L})_{pre} \tag{6.1}$$

The quantities on the right hand side (RHS) of the equation are calculated from

$$s_a(\mathcal{L})_{pre/post} \equiv \sum_i s_a(\ell_i)_{pre/post} \tag{6.2}$$

where ℓ_i represents the i^{th} change set in location $\mathcal{L}$. The metric for the total change in static afferent coupling in the system is then given by

$$\Delta s_{a_{tot}} = \Delta s_a(B) + \Delta s_a(N) + \Delta s_a(F) \tag{6.3}$$

6.0.2.2 Static Efferent Coupling Change Metrics

The static efferent coupling metric is denoted by s_e and parametrized with the location as done for s_a. The metric for a single location is defined as

$$\Delta s_e(\mathcal{L}) = s_e(\mathcal{L})_{post} - s_e(\mathcal{L})_{pre} \tag{6.4}$$

The quantities on the RHS are calculated from

$$s_e(\mathcal{L})_{pre/post} \equiv \sum_i s_e(\ell_i)_{pre/post} \tag{6.5}$$

where ℓ_i represents the i^{th} change set in location $\mathcal{L}$. The metric for the total change in static afferent coupling in the system is then given by

$$\Delta s_{e_{tot}} = \Delta s_e(B) + \Delta s_e(N) + \Delta s_e(F) \tag{6.6}$$

6.0.2.3 Dynamic Afferent Coupling Change Metrics

The dynamic afferent coupling metric is denoted by d_a and parametrized with the location as previously done. The metric for a single location is defined as

$$\Delta d_a(\mathcal{L}) = d_a(\mathcal{L})_{post} - d_a(\mathcal{L})_{pre}$$

The quantity on the RHS is calculated from

$$d_a(\mathcal{L})_{post} \equiv \sum_i d_a(\ell_i)_{post} \tag{6.7}$$

where ℓ_i represents the i^{th} change set in location $\mathcal{L}$. The metric for the total change in static afferent coupling in the system is then given by

$$\Delta d_{a_{tot}} = \Delta d_a(B) + \Delta d_a(N) + \Delta d_a(F) \tag{6.8}$$

6.0.2.4 Dynamic Efferent Coupling Change Metrics

The dynamic efferent coupling metric is denoted by d_e and parametrized with the location as previously done. The metric for a single location is defined as

$$\Delta d_e(\mathcal{L}) = d_e(\mathcal{L})_{post} - d_e(\mathcal{L})_{pre}$$

The quantities on the RHS are calculated from

$$d_e(\mathcal{L})_{post} \equiv \sum_i d_e(\ell_i)_{post} \tag{6.9}$$

where ℓ_i represents the i^{th} change set in location $\mathcal{L}$. The metric for the total change in static afferent coupling in the system is then given by

$$\Delta d_{e_{tot}} = \Delta d_e(B) + \Delta d_e(N) + \Delta d_e(F) \tag{6.10}$$

6.0.3 INTERNAL COMPLEXITY METRICS

Ideally, we would like to be able to integrate new functionality without having to make modifications to existing code. Some architectures help in getting close to that ideal while other architectures do nothing and in some cases seem to hinder the effort. When undertaking an integration effort, an early understanding of how much effort will be needed to perform the integration can help provide scope and realistic expectations on the timing and result of the effort. Most commercial practices around software estimation (still) rely on LOC metrics to give rough estimates of task level of effort. The LOC metrics are evaluated against organizational norms for developer efficiency and from this evaluation an initial estimate of both effort required and associated economic cost may be made. To compare the integration task level of effort between different architectures we define metrics that capture an implied cost for change within an architecture due to complexity. The interpretation of such a metric is that architectures with a high change cost are less desirable than those with a lower change cost.

We begin by defining a line of code which will be the basis for our level of effort estimate. A line of code is any executable statement in the implementation language of the element under consideration. This definition includes data definitions, type definitions, and lines with implied executability such as the **switch/case** statement in C-like languages. The definition also includes language constructs such as imports and includes but excludes lines such as include guards typically found in C or C++. Additionally, the function signature line for the definition of a function is included since many languages permit function arguments to either have default values (assignment) or do work such as call out to another function. However, we only count the function signature line as a single LOC. Finally, source lines continued across multiple physical lines as viewed in an editor are counted as a single LOC, i.e., we will count source line termination symbols, e.g., ";" in C-like languages, as defined in the implementation language. This choice also allows us to count multiple source lines on a single line as more than a single LOC, e.g., *for* statements in C or C++.

6.0.3.1 Change Counts

Modifying source code consists of three possible actions: addition of lines, removal of lines, or modification of a line. The modification action is not detectable from simple counting of the source code since it neither increases nor decreases the total LOC. However, we are less interested in the actual count of LOC and more interested in the count of changed lines; to accurately capture change count, modified lines must be included. Counting modified lines requires access to either the unchanged and changed integration source for a line-by-line comparison or to version control logs where all modifications are recorded. The change in LOC in an element due to integration work is denoted by Γ_{tot} and is defined as

$$\Gamma_{tot} = \Gamma_A + \Gamma_R + \Gamma_M \tag{6.11}$$

where Γ_A is the total number of lines added, Γ_R is the total number of lines removed, and Γ_M is the total number of lines modified. These quantities are defined as

$$\Gamma_A = \Gamma_A(B) + \Gamma_A(N) + \Gamma_A(F) \tag{6.12}$$

$$\Gamma_R = \Gamma_R(B) + \Gamma_R(N) + \Gamma_R(F) \tag{6.13}$$

$$\Gamma_M = \Gamma_M(B) + \Gamma_M(N) + \Gamma_M(F) \tag{6.14}$$

where the B, N, and F represent base, new, and framework as previously defined. We can define the change counts in the base, new, and framework locations in the same way as we did for the coupling metrics. Thus,

$$\Gamma(\mathcal{L})_{tot} = \Gamma_A(\mathcal{L}) + \Gamma_R(\mathcal{L}) + \Gamma_M(\mathcal{L}) \tag{6.15}$$

$$\Gamma_A(\mathcal{L}) = \sum_i \Gamma_A(\ell_i) \tag{6.16}$$

$$\Gamma_R(\mathcal{L}) = \sum_i \Gamma_R(\ell_i) \tag{6.17}$$

$$\Gamma_M(\mathcal{L}) = \sum_i \Gamma_M(\ell_i) \tag{6.18}$$

With the change count metrics defined, we have estimates[2] of the effort required to perform the integration. However, because we have included modified lines and defined the change counts without reference to the unchanged and changed overall code size, we have lost information about the impact of the code change. We make the assumption that small changes in a large body of code are likely to have less of an impact to the overall complexity of the code than the same changes to a small body of code. To recapture this information, we scale the change count by the ratio of post-integration LOC to pre-integration LOC. This ratio captures the relative size change of the body of code and adjusts the total line change accordingly. To simplify the notation, denote the lines of code count by λ. The scale factor we use is

$$\hat{\lambda} = \frac{\lambda_{post}}{\lambda_{pre}} \tag{6.19}$$

Accounting for location dependence gives

$$\hat{\lambda}(\mathcal{L}) = \frac{\lambda(\mathcal{L})_{post}}{\lambda(\mathcal{L})_{pre}} \tag{6.20}$$

With these scale factors, we can rewrite the change counts as scaled change counts, denoted by $\bar{\Gamma}$, and obtain

$$\bar{\Gamma}_{tot} = \hat{\lambda} \cdot \Gamma_{tot} \tag{6.21}$$

$$\bar{\Gamma}_A = \hat{\lambda} \cdot \Gamma_A \tag{6.22}$$

$$\bar{\Gamma}_R = \hat{\lambda} \cdot \Gamma_R \tag{6.23}$$

$$\bar{\Gamma}_M = \hat{\lambda} \cdot \Gamma_M \tag{6.24}$$

With location dependence included, we obtain

$$\bar{\Gamma}(\mathcal{L})_{tot} = \hat{\lambda} \cdot \Gamma(\mathcal{L}) \tag{6.25}$$

[2]We call this an estimate because this calculation can in practice be done prior to the actual work of the integration during the planning stages.

$$\bar{\Gamma}_A(\mathcal{L}) = \hat{\lambda}(\mathcal{L}) \cdot \Gamma_A(\mathcal{L}) \tag{6.26}$$

$$\bar{\Gamma}_R(\mathcal{L}) = \hat{\lambda}(\mathcal{L}) \cdot \Gamma_R(\mathcal{L}) \tag{6.27}$$

$$\bar{\Gamma}_M(\mathcal{L}) = \hat{\lambda}(\mathcal{L}) \cdot \Gamma_M(\mathcal{L}) \tag{6.28}$$

6.0.3.2 Complexity Change Cost Metrics

With change count metrics we have an idea of the effort required to perform an integration and this estimate can be used in comparing architectures for fitness. However, there is more impact from the code changes than just increasing or decreasing the size of the source code. The additional impact is in the form of changes to the internal complexity of the elements induced by the code changes. To capture this change and to provide a means of comparison between architectures, we define a complexity change cost metric, E, as the effort required to change the complexity by one unit (based on whatever complexity measure is chosen). We will use the McCabe Cyclomatic Complexity, v_g, as our measure of complexity and so a unit change in complexity represents the addition of a new independent path in the source code. This is important because such increases represent additional effort in understanding, testing, and validation. The complexity change is then

$$\Delta v_g = v_{g_{post}} - v_{g_{pre}} \tag{6.29}$$

Accounting for location dependence, we obtain

$$\Delta v_g(\mathcal{L}) = v_g(\mathcal{L})_{post} - v_g(\mathcal{L})_{pre} \tag{6.30}$$

with the decomposition of $v_g(\mathcal{L})_{post}$ and $v_g(\mathcal{L})_{pre}$ by element given by

$$v_g(\mathcal{L})_{pre/post} = \sum_i v_g(\ell_i)_{pre/post} \tag{6.31}$$

Similar to the change counts, the complexity change provides insufficient information to make a determination on the real impact of the complexity change, so we make

the same assumption as in the change in LOC and scale the complexity change value in the same way as the scaled change counts. The scale factor is

$$\hat{v}_g = \frac{v_{g_{post}}}{v_{g_{pre}}} \tag{6.32}$$

and with location dependence becomes

$$\hat{v}_g(\mathcal{L}) = \frac{v_g(\mathcal{L})_{post}}{v_g(\mathcal{L})_{pre}} \tag{6.33}$$

With these scale factors, the complexity change in Equations (6.29) and (6.30) becomes

$$\Delta\bar{v}_g = \hat{v}_g \cdot \Delta v_g \tag{6.34}$$

$$\Delta\bar{v}_g(\mathcal{L}) = \hat{v}_g(\mathcal{L}) \cdot \Delta v_g(\mathcal{L}) \tag{6.35}$$

With the complexity change defined in Equations (6.34) and (6.35) and change counts defined by Equations (6.21) and (6.25), we can now define the complexity change cost metrics, E and $E(\mathcal{L})$ as

$$E = \frac{\bar{\Gamma}_{tot}}{\Delta\bar{v}_g} \tag{6.36}$$

$$E(\mathcal{L}) = \frac{\bar{\Gamma}_{tot}(\mathcal{L})}{\Delta\bar{v}_g(\mathcal{L})} \tag{6.37}$$

The denominators in Equations (6.36) and (6.37) can potentially be zero, so to avoid the singularity but still obtain a meaningful result we impose the constraints

$$E = 0 \quad \text{if} \quad \Delta\bar{v}_g = 0 \tag{6.38}$$

and

$$E(\mathcal{L}) = 0 \quad \text{if} \quad \Delta\bar{v}_g(\mathcal{L}) = 0 \tag{6.39}$$

These constraints state that if no complexity change was measured, then the effort of changing the code is negligible and we assign zero to the metric.

6.0.4 USING THE METRICS

The metrics defined above are built from metrics computed at some lower level of software element structure. The granularity of this level will depend upon the fundamental unit of structure within the implementation language and the architecture being evaluated. For example, computing the afferent coupling of objects can be done at the object level since the object is the fundamental structural unit within object oriented programming languages. For components, however, the afferent coupling is computed at the level of the component. This level is chosen because for software component architectures, the component, while generally consisting of interacting objects, is considered fundamental and any use of the component should not depend upon the internal structure of the component. When computing the internal complexity of software elements, however, we build up the complexity metrics as the sum of the complexities of the element parts. Since the complexity change cost is computed based on the effort as measured through changed lines of code, the measurements must be taken at the LOC level. This level choice is consistent with our identification of these metrics as "internal" complexity metrics. Effort metrics other than lines of code and complexity metrics other than v_g could be substituted in the above calculations with the appropriate change of units, but when comparing different architectures it is important to compare the complexity change cost computed using the same effort and complexity metrics for each architecture.

In general, interpretation of the coupling metrics is made by associating a high degree of coupling with a more complex structure. This is done because elements with many connections are considered to potentially be doing too much work. For object oriented systems, this leads to identification of poorly designed systems. However, in some architectures, a single element may be highly coupled but serves a single purpose. For architectures based on messaging strategies, the message broker element will exhibit very high dynamic coupling because all messages must flow through the broker. In these cases, we consider the message broker to be a single purpose element with low complexity despite the high coupling. Thus, when interpreting the coupling

metrics defined above, care must be taken to ensure that the role of the element is considered as a part of the interpretation. Part of our use of the location parametrization is to account for this role effect. We believe that complexity in the framework elements is preferable to complexity in the base or new code locations. However, that preference is based on the notion that framework elements should hide complexity from users of the framework and the degree to which that hiding is done affects the perceived complexity of the framework.

The complexity change cost metrics are interpreted by assigning lower values of the metric with an architecture that is more acceptable or fit. This interpretation is made based on the idea that integration inherently increases the complexity of the system through the addition of new capabilities. Because of this unavoidable complexity increase, integration done within architectures that require more effort to change the complexity, as measured by the LOC change count, is more difficult and more costly in terms of people, time, and money.

The metrics as defined would appear to require a fully developed system to make measurements against. In fact, better results are obtained from measurement of the source after the integration is complete. However, the metrics have the potentail to be used predictively, but with reduced accuracy. The base, framework, and new source will already exist since otherwise we would not be considering this an integration. Examination of the existing source is a precursor to any integration effort and organizations adept at integration will create an integration strategy and integration plan based on that examination. Our metrics can be computed by first creating an integration plan for each capability to be integrated and for each architecture to be considered and use the results of those plans to make the necessary measurements. This use will be an estimate only but should provide sufficient insight into how different architectures compare when doing the same integration tasks without having to actually perform the integration.

Coupling was defined in the context of object oriented systems. Since some of the code we may need to analyze will not be object oriented, we need some guidelines for determining coupling in those implementations. For example, since C is not object-

oriented, the software tools available for measuring coupling do not analyze C code [Chidamber and Kemerer, 1994]. To extract coupling measures for the C portions of the integrations, we consider the use of include files (.h) other than stdio.h and stdlib.h as an indication of static coupling. Include files distributed with applications written in C typically encapsulate datatype definitions and operations on those types in such a way that the individual files can be considered as a form of data type partitions. We used this partitioning to provide a rough measure of coupling for all applications written in C by counting the number of include files referenced in the application.

Measuring dynamic coupling is difficult in general. Profiling tools give an indication of dynamic coupling but are only applicable to code compiled or instrumented with profiling enabled for the specific profiling tool. To get a more accurate count of dynamic coupling, we scanned the source code for occurrences of linkages between types or classes that could not exist except during execution of the application. These linkages included

- Dynamic library loads,
- Dynamic method invocation (in the case of Ice),
- *accept*, *bind*, *connect*, *send*, *recv*, *read*, or *write* operations on sockets, and
- Any usage of runtime type information (RTTI).
- Linkages that could be modified through configuration information changes without requiring recompilation

For each occurrence of these constructs, the dynamic coupling count is incremented by one.

In Parts III, IV, and V we use our metrics to evaluate how well the service oriented approach to integration compares to the approaches based on Player and P2P. For all of the integration tasks performed, the integration was performed in stages by creating an operational baseline that was then used to measure how much

change occurred when the new capabilities were integrated in small increments. The operational baseline is termed "baseline" and measurements taken on the baseline establish how much effort was required to get a functional capability to support further integration. The remaining stages are termed "first" and "second." These represent the addition of a single new piece of functionality to the operational baseline at each stage. Measurements are made at each stage and reported by stage.

Part III

Comparison of World Representation Database Integration Approaches

7. WORLD DATABASE INTEGRATION

In this chapter we provide a discussion of the implementation details for integrating the World Database. In addition, we provide the detailed measured counts of code lines, McCabe complexity, and computed values of the various metrics used in comparing architectural approaches to integration. The data values are grouped by the integration phase in which they were collected and further grouped and summarized by the location of the software element from which the value was obtained. Locations are either Framework (F), Base (B), or New (N).

7.1 SOA Integration

For the SOA approach to the WDB task, the following classes and applications were created in addition to those components provided by the ZeroC Ice™ and ZeroC IceStorm™ products:

- ATRV_WDB_Impl—Implementation class of the defined interface. Provides the connection to the Ice runtime components through inheritance of generated classes containing Ice-specific implementations. For the purposes of integration, this class was implemented as two separate classes, ATRV_WDB_Impl_C and ATRV_WDB_Impl_S, for use in client and server applications, respectively.

- ServiceUtilities—Collection of utilities wrapping calls to the communications adapter interface. Provides a decoupling of the message exchange patterns from the communications transports. Also contains utilities for manipulating XML files.

- MURI_DataRep—Class containing the API used by the client application. Uses the ServiceUtilities class to perform messaging but manages the appearance of the message flow for the client. For the WDB task, the client expects

synchronous method invocations, so the MURI_DataRep class provides blocking calls on the API methods but uses asynchronous messaging through the interface definition and ZeroC IceStorm™ .

- IceStormAdapter—Implementation class of the SOA_Base interface. SOA_Base defines generic functionality associated with typical communication protocols. IceStormAdapter implements that interface using the ZeroC IceStorm™ messaging product.

- DataServiceMain—Server application built on top of ServiceUtilities and IceStormAdapter to receive published requests for service from the client API calls for database access. Translates those request messages into the correct database access executable. Also responsible for delivering results XML files as a reply message to the client request.

- CPPTestClient—Simple client used to test the functionality provided by the other components and demonstrate usage.

All measurements were made on these classes and are reported at the class level. Additionally, no measurements were made on code generated by Ice or on any library code provided by Ice. The next section presents the computed metrics by the code location, i.e., either Framework, New, or Base. The detailed measurement data can be found in Appendix C.

7.2 Player Integration

The code created during the Player integration of the World Database is divided into three components. The WDB functionality is delivered through calls into the driver component. Those calls are delivered from the client indirectly through the interface client proxy. In all cases, the calls are mediated by the Player server provided by the Player framework. The three components are as follows:

- wdb_client—C language implementation of a simple client for testing the messaging interfaces,

- wdbinterf_client—C language implementation of the interface defined for the WDB tasks, and
- wdbinterf_driver—C++ language implementation of the driver providing access to the WDB query and update executables.

Detailed measurement data and computation of the metrics can be found in Appendix D

7.3 P2P Integration

The P2P approach is the simplest in terms of code division, consisting of a server application and a client application. The code is contained in two C language files as follows:

- wdb-client—Client code for connecting through internet sockets to the server on a well-known port and for streaming the request data across the socket to the server, and
- wdb-server—Server code for listening for socket connections, receiving data from clients, executing the WDB query and update executables, and streaming results across the socket to the client.

Detailed measurement data and computation of the metrics can be found in Appendix E

7.4 Metrics Results

In this section, we present the results of the measurements and metrics computations for the different integration approaches used in the WDB integration task. The first three graphs depict the results of the coupling measurements. The results are grouped by the type of the coupling and the values by location are shown. The remaining three graphs depict the results of measuring code size change, complexity

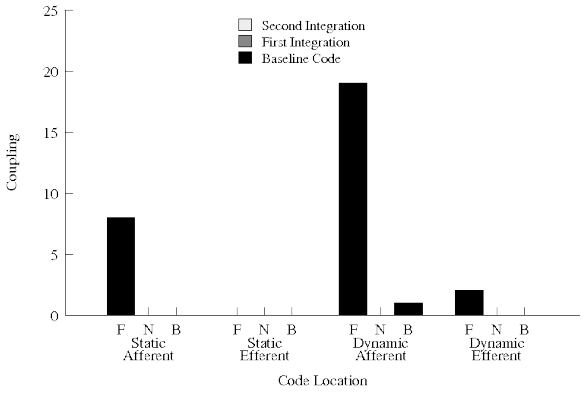

Figure 7.1: Coupling measurements from the SOA approach to the WDB integration task. Detailed data for this graph can be found in Appendix C, Tables C.4, C.10, and C.16.

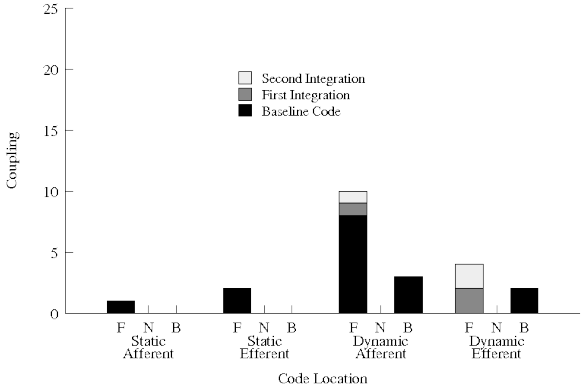

Figure 7.2: Coupling measurements from the Player approach to the WDB integration task. Detailed data for this graph can be found in Appendix D, Tables D.4, D.10, and D.16.

change, and complexity change cost for the different integration methods. The results are grouped by integration approach and the values by location are shown.

The measured coupling data for the different integration approaches is shown in Figures 7.1, 7.2, and 7.3.

The scaled code size metric data for the different integration approaches is shown in Figure 7.4. The data are clustered by integration approach. Within each cluster, the values are separated by location.

The scaled complexity metric data for the different integration approach is shown in Figure 7.5. The data are clustered by integration approach. Within each cluster, the values are separated by location.

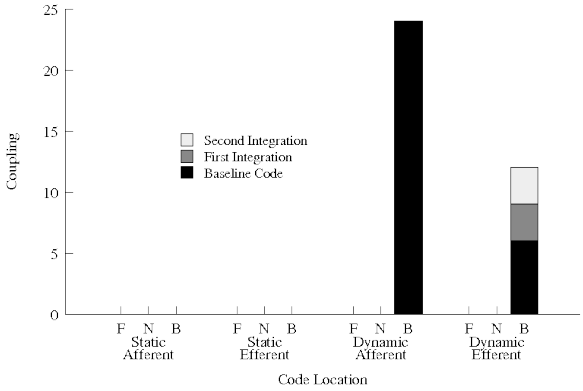

Figure 7.3: Coupling measurements from the P2P approach to the WDB integration task. Detailed data for this graph can be found in Appendix E, Tables E.4, E.10, and E.16.

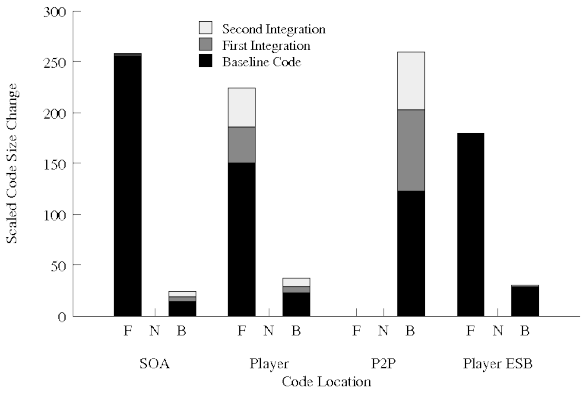

Figure 7.4: Scaled code size measurements from the various integration approaches to the WDB integration task. Detailed data for this graph can be found in Appendix C, Tables C.5, C.11, and C.17, Appendix D, Tables D.5, D.11, and D.17, and Appendix E, Tables E.5, E.11, and E.17.

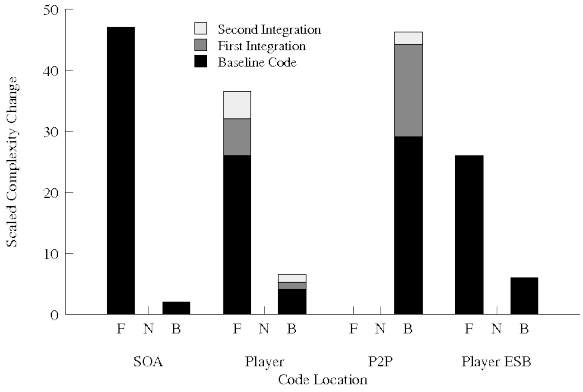

Figure 7.5: Scaled complexity metrics from the various integration approaches to the WDB integration task. Detailed data for this graph can be found in Appendix C, Tables C.6, C.12, and C.18, Appendix D, Tables D.6, D.12, and D.18, and Appendix E, Tables E.6, E.12, and E.18.

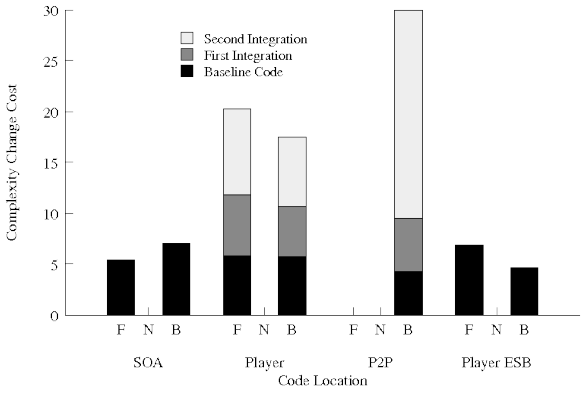

Figure 7.6: Complexity Change Cost metric values from the various integration approaches to the WDB integration task. Detailed data for this graph can be found in Appendix C, Tables C.6, C.12, and C.18, Appendix D, Tables D.6, D.12, and D.18, and Appendix E, Tables E.6, E.12, and E.18.

The Complexity Change Metric metric for the different integration approaches is shown in Figure 7.6. The data are clustered by integration approach. Within each cluster, the values are separated by location.

8. COMPARATIVE RESULTS

In the previous chapter we presented the details of the measurements and metrics derived from the staged integrations performed for the World Database integration tasks. In this chapter we will summarize those results and provide an interpretation of the summary values.

8.1 Coupling Metrics

In this section we consider the coupling between components and how integration is affected by that coupling and how the coupling is changed during integration. The summary metrics for the coupling changes are shown in Figure 8.1. In the figure, the numbers correspond to the SOA, Player, P2P, and Player as ESB integration approaches, respectively. In the SOA approach for WDB integration, the coupling does not change. Several factors contribute to this lack of change including

- The use of generic Request and Reply operations,
- The use of the Context object to transport the database operation names,
- The arguments to both operations are strings representing the contents of the XML request and result files, and
- Client code accesses the services by calling *execute(op-name)*.

Thus, addition of a new database operation does not change anything except the values of the strings used in the service request.

For the Player integration in the WDB integration, the change in efferent coupling is constant for both static and dynamic coupling. The lack of variability in the efferent coupling can be understood by considering that addition of a new

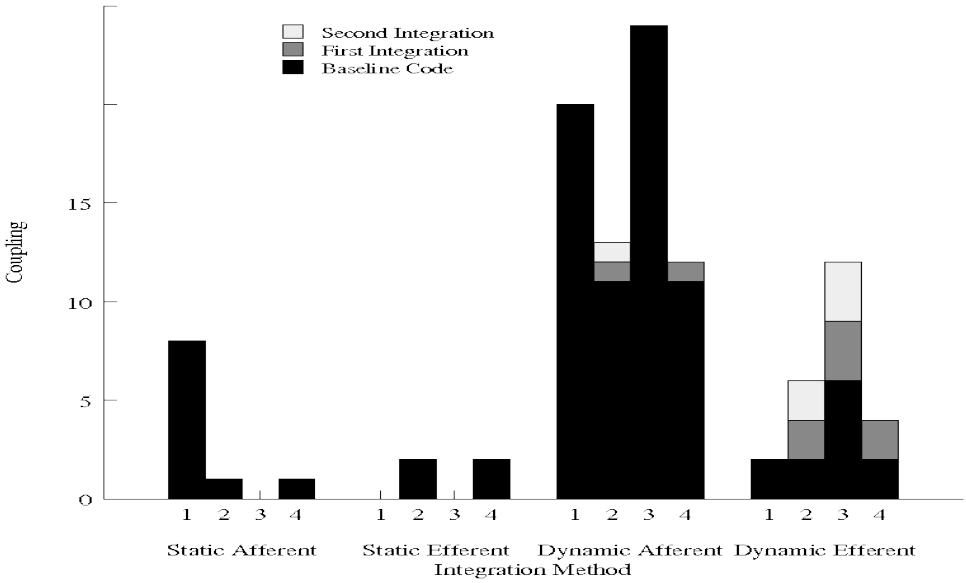

Figure 8.1: Summary of coupling changes for the various integration approaches to the WDB task

message type to Player results in changes to the message handling code for the new message resulting in the +1 increase in static efferent coupling. Additionally, the new message contributes a new interface implementation in the client proxy and the driver code, both of which are accessed at runtime, resulting in the +2 increase in dynamic efferent coupling.

In the P2P approach to the WDB integration, each new message adds an *accept*, *send*, and *recv* operation for handling the request and response. Since our guidelines count these as dynamic linkages, we obtain the constant +3 increase in dynamic efferent coupling at each integration step.

8.2 Complexity Change Cost

In this section we present the summary metrics related to complexity changes due to integration. The summary values for LOC changes, McCabe complexity changes, and the resulting complexity change cost are shown in Figures 8.2, 8.3, and 8.4. In the SOA approach for the WDB integration, we encounter a high complexity change cost in the initial creation of the baseline capability. However, the large up-front cost pays off in subsequent integrations where the complexity change

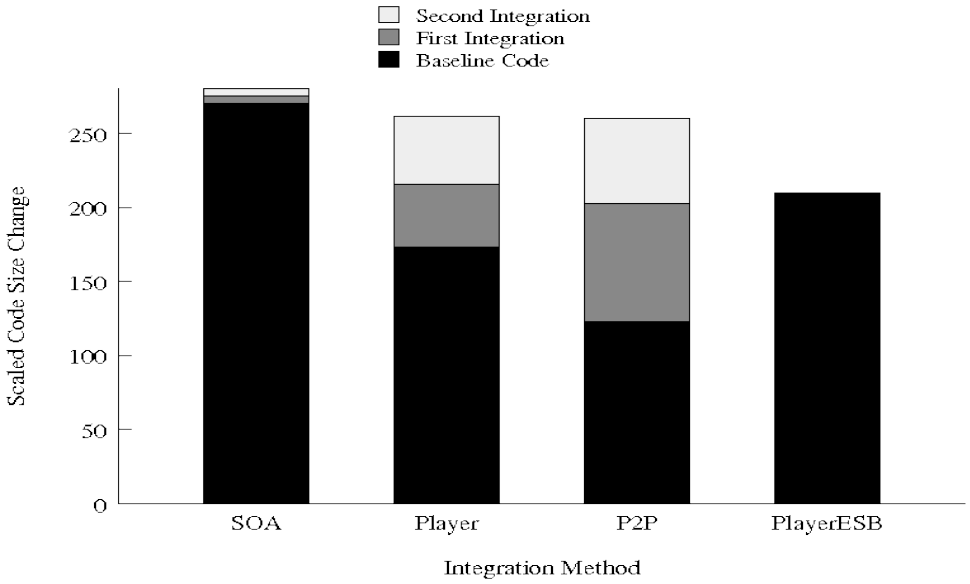

Figure 8.2: Code volume changes across the baseline, first, and second integration stages for the WDB task

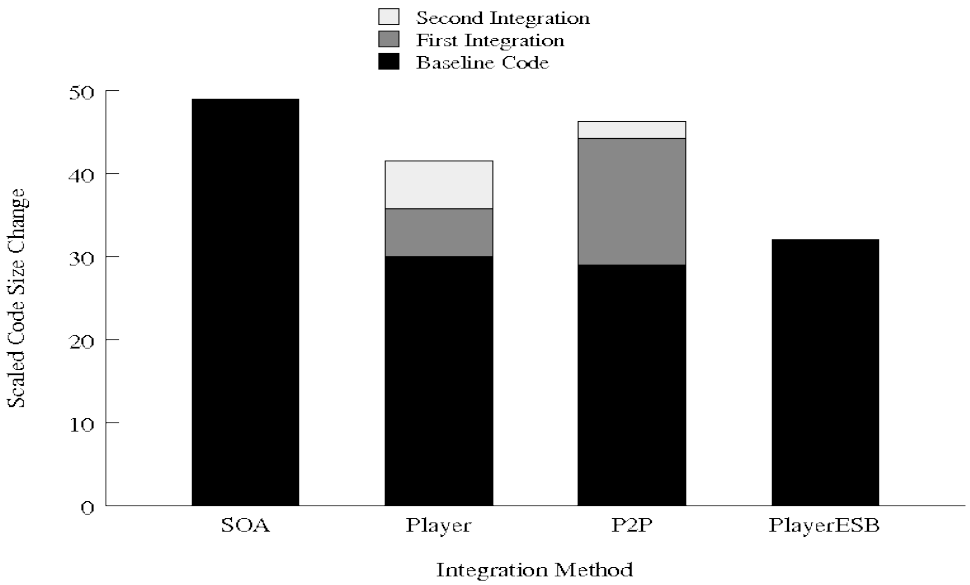

Figure 8.3: Complexity changes across the baseline, first, and second integration stages for the WDB task

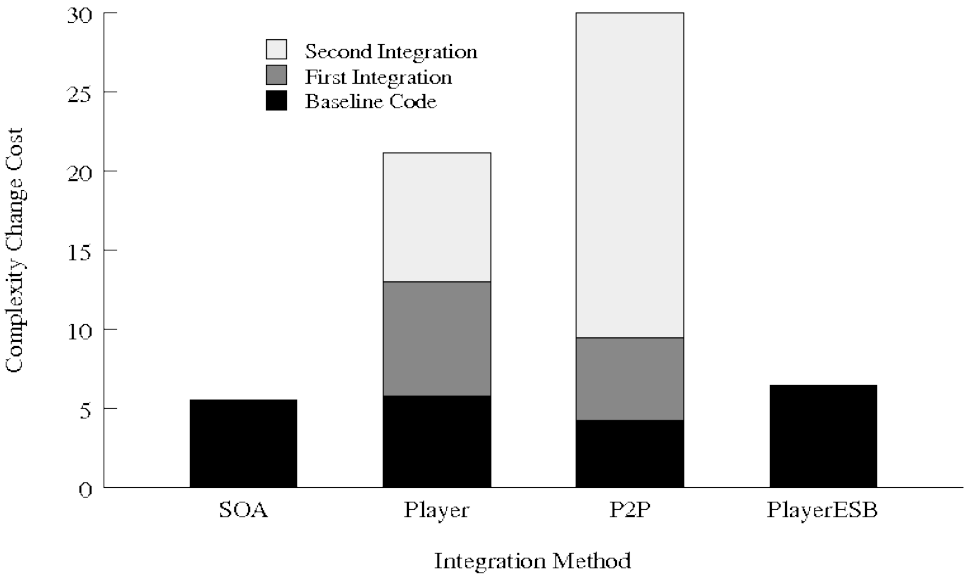

Figure 8.4: Complexity change cost across the baseline, first, and second integration stages for the WDB task

cost is zero. Adding a new capability using the ESB framework consists of the addition of two lines in the client (one to call the service, the other to read the results) and a single line in the service provider code to dispatch the request. These changes are sequential lines of code and thus do not increase the complexity of the source, resulting in a zero complexity change cost after the initial effort.

For the Player approach to the WDB integration, the baseline codebase is relatively small but with high complexity. With successive integrations, the source code grows by 30 to 40 lines with a scaled complexity of around 6. The nearly constant change in complexity is due to the need to perform checks on messages at various points to understand what message has been received and what should be done with that message. These checks are the same for each message except for the defined constant associated with a particular message. This consistency is what gives a constant growth with each message addition. However, since the complexity change cost is the ratio of scaled LOC changes to scaled complexity, the complexity change cost will continue to grow as more capabilities are added because of the way in which those capabilities must be added.

In the case of P2P for WDB integration, the size of the source code and com-

plexity grow significantly with each integration. Since the socket operations used to transmit and receive messages are very low-level constructs, significant error checking and handling code is required to handle both incoming and outgoing data. Additionally, once a second message has been added to the P2P interactions, decisions about protocols for embedding function returns and parameter values and types must be made. These decisions require additional code to handle message transformations to and from the different protocols, thus increasing the size and complexity of the source code. In short, P2P integration requires creation of capabilities that exist in frameworks such as Player or an ESB.

Part IV

Comparison of SUBTLE Pragmatics Integration Approaches

9. PRAGMATICS INTEGRATION WITH SOA

In this chapter we provide details of the service implementation for the MURI Pragmatics integration tasks. In addition, the detailed measured counts of code lines, McCabe complexity, and computed values of the various metrics are given. The data values are grouped by the integration phase in which they were collected and further grouped and summarized by the location of the software element from which the value was obtained. Locations are either Framework (F), Base (B), or New (N).

9.1 SOA Integration

For the Pragmatics integration task using the SOA approach, the solution consists of three distinct components: the GUI, Pragmatics, and the Backend component. The interface definition used in the three components consists of three request operations and one reply operation. The three request operations differ only in the argument list taken by each and support the three different argument lists used in the overall Pragmatics system. The reply operation is not used but is included to allow for the addition of request-response type messaging. The following classes and applications were created to implement these components:

- Backend_Messages—Class presenting the backend message API. This class allows application code in the Backend to send status update messages to the GUI through the Pragmatics component,
- Backend_PragImpl—Backend implementation class of the interface,
- Prag_Messages—Class presenting the Pragmatics component API. This class allows application code in the Pragmatics component to remap messages and transfer data sent from the other components,

- Prag_PragmaticsImpl—Pragmatics component implementation class of the interface,
- GUI_Messages—Class presenting the GUI component API. This class allows application code in the GUI to send commands to the Backend,
- GUI_PragmaticsImpl—GUI component implementation class of the interface,
- ServiceUtilities—Collection of utilities for handling initialization of the communications channels and setup and sending of the IceStorm messages between the components. This class is similar to the ServiceUtilities class in the WDB integration but was refactored and amended to accommodate the differing needs of the Pragmatics integration. Future versions of this class should not require amendment to be used across integration tasks with differing requirements,
- IceStormAdapter—Communications transport adapter class implementing the SOA_Base interface. This adapter class is the same as that used in the WDB integration,
- gui—C++ language application to initiate messaging between the GUI and the Pragmatics component,
- prag—C++ language application to handle calls from the GUI, deliver commands to the Backend, and transfer status updates from the Backend to the GUI, and
- backend—C++ language application to execute the backend functionality based on messages received from the Pragmatics component.

As in the WDB task previously described, we do not analyze libraries provided by the Ice or IceStorm product nor do we analyze code generated by the Ice IDL compiler. The detailed measurement data can be found in Appendix F.

9.2 Player Integration

Similar to the SOA approach, three distinct components are identified as the GUI, Pragmatics, and the Backend. For a Player integration, we must create a client proxy and a driver for each of the components. Additionally, client functionality to access the functionality was created. In the case of the Pragmatics component, this functionality was a hard-wired mapping of GUI commands to Backend commands due to the lack of an actual Pragmatics capability at the time of the integration effort. This lack of functionality does not create difficulties with the integration since the code in place is easily replaceable with the Pragmatics functionality as described to us by the creators of that functionality. Based on this, we created nine separate code packages as follows:

- muribackendinterf_client—Client proxy C code for the Backend component messages,
- muribackendinterf_driver—Driver C++ code for the Backend device,
- muripraginterf_client—Client proxy C code for the Pragmatics component messages,
- muripraginterf_driver—Driver C++ code for the Pragmatics device,
- muriguiinterf_client—Client proxy C code for the GUI component messages,
- muriguiinterf_driver—Driver C++ code for the GUI device,
- muri_backend—C code application used to access the Backend component functionality, a wrapper around commands received from the Pragmatics component,
- muri_pragmatics—C code application used to create the link between the GUI component and the Backend component. Primarily responsible for mapping GUI command into equivalent Backend commands and vice versa, and

- muri_gui—C code application used to provide commands from the GUI to the Backend component through the Pragmatics component.

Detailed measurement data can be found in Appendix G.

9.3 P2P Integration

As in the SOA and Player implementations, the GUI, Pragmatics, and Backend components formed the basic high level structure of the implementation. Because of the nature of the P2P approach, we provided the implementation of each component in a single file, i.e., one per component. This means that since each component functioned as both client and server, the code for each role is contained in the same file. The code for each component is thus divided into the following three files:

- muri-backend—Client and server code for the backend functionality,
- muri-prag— Client and server code for the pragmatics functionality, and
- gui-prag—Client and server code for the GUI functionality.

The detailed measurement data can be found in Appendix H.

9.4 Metrics Results

In this section, we present the results of the measurements and metrics computations for the different integration approaches used in the WDB integration task. The first three graphs depict the results of the coupling measurements. The results are grouped by the type of the coupling and the values by location are shown. The remaining three graphs depict the results of measuring code size change, complexity change, and complexity change cost for the different integration methods. The results are grouped by integration approach and the values by location are shown.

The measured coupling data for the different integration approaches is shown in Figures 9.1, 9.2, and 9.3.

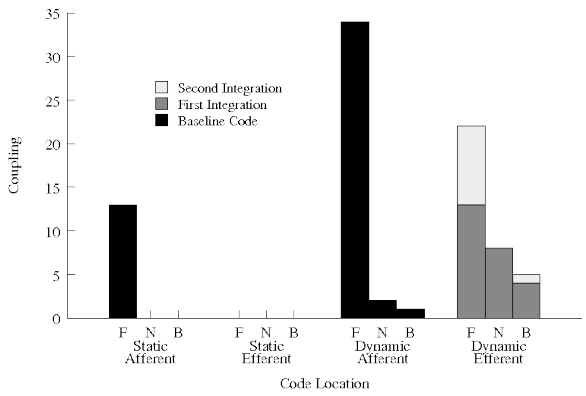

Figure 9.1: Coupling measurements from the SOA approach to the Pragmatics integration task. Detailed data for the graph can be found in Appendix F, Tables F.4, F.10, and F.16.

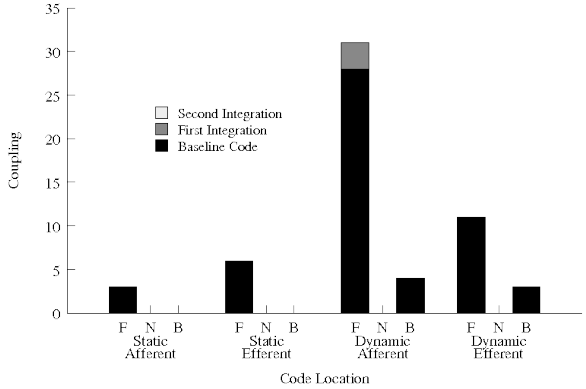

Figure 9.2: Coupling measurements from the Player approach to the Pragmatics integration task. Detailed data for the graph can be found in Appendix G, Tables G.4, G.10, and G.16.

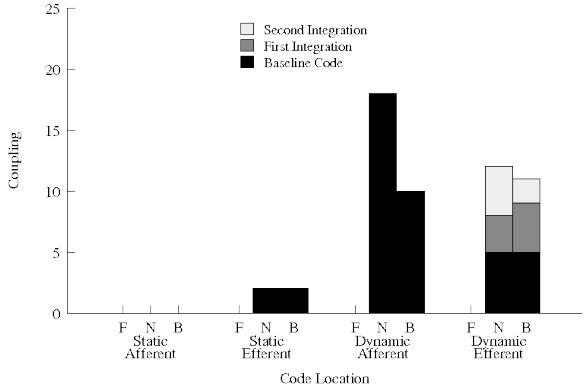

Figure 9.3: Coupling measurements from the P2P approach to the Pragmatics integration task. Detailed data for the graph can be found in Appendix H, Tables H.4, H.10, and H.16.

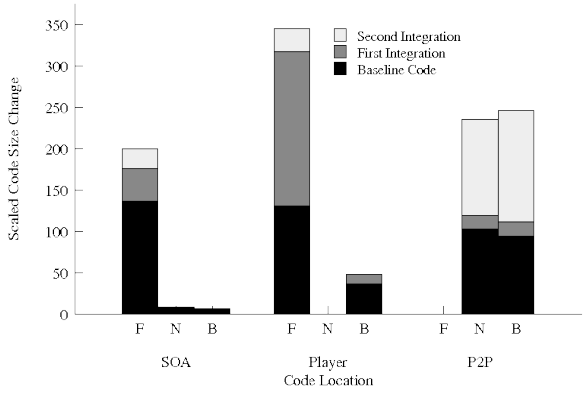

Figure 9.4: Scaled code size measurements from the various integration approaches to the Pragmatics integration task. Detailed data for the graph can be found in Appendix F, Tables F.5, F.11, and F.17, Appendix G, Tables G.5, G.11, and G.17, and Appendix H, Tables H.5, H.11, and H.17.

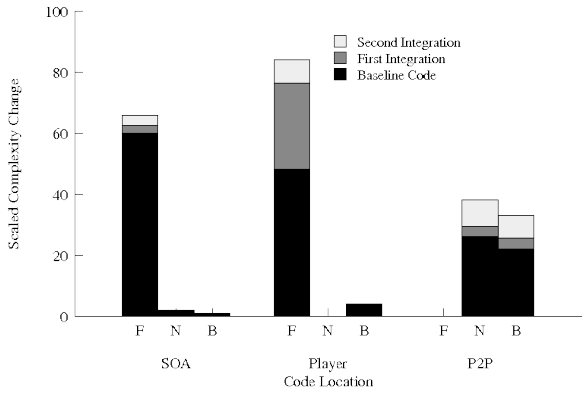

Figure 9.5: Scaled complexity metrics from the various integration approaches to the Pragmatics integration task. Detailed data for the graph can be found in Appendix F, Tables F.6, F.12, and F.18, Appendix G, Tables G.6, G.12, and G.18, and Appendix H, Tables H.6, H.12, and H.18.

The scaled code size metric data for the different integration approaches is shown in Figure 9.4. The data are clustered by integration approach and the values are broken out by location.

The scaled complexity metric data for the different integration approach is shown in Figure 9.5. The data are clustered by integration approach and the values are broken out by location.

The Complexity Change Metric metric for the different integration approaches is shown in Figure 9.6.

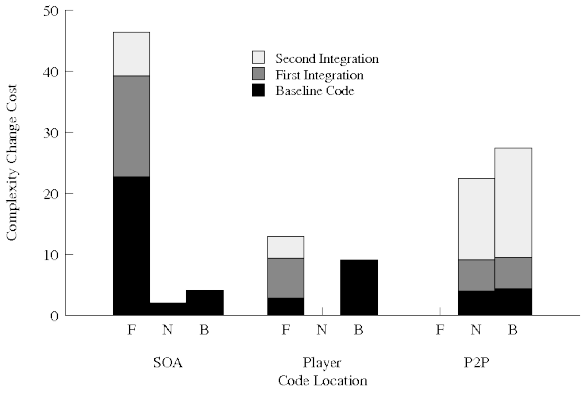

Figure 9.6: Complexity Change Cost metric values from the various integration approaches to the Pragmatics integration task. Detailed data for the graph can be found in Appendix F, Tables F.6, F.12, and F.18, Appendix G, Tables G.6, G.12, and G.18, and Appendix H, Tables H.6, H.12, and H.18.

10. COMPARATIVE RESULTS

In the previous chapter we presented the details of the measurements and metrics derived from the staged integrations performed for the SUBTLE integration tasks. In this chapter we will summarize those results and provide an interpretation of those summary values.

10.1 Coupling Metrics

In this section we consider the coupling between components and how integration is affected by that coupling and how the coupling is changed during integration. The summary coupling metrics are shown in Figure 10.1.

For the SOA approach, the dynamic efferent coupling is relatively large for both integration phases. This large value is due to the way the commands and status updates are dynamically created in the messaging component, and the effect that the dynamic creation has on how the interface implementations must deal with the results. The large value is also due to small contributions of a messaging component and interface implementation for each of the GUI, Pragmatics, and the Backend.

In the Player approach, the initial coupling change from the baseline increases due to the addition of a new message in each of the GUI, Pragmatics, and Backend. Otherwise, the coupling is unchanged since the addition of new messages does not induce any additional coupling above that previously described in Chapter 8

Finally, using the P2P approach we see an increase in dynamic efferent coupling due to the addition of *accept*, *recv*, and *send* operations for each of the GUI, Pragmatics, and Backend components similar to that observed in the World Database task described in Chapter 8.

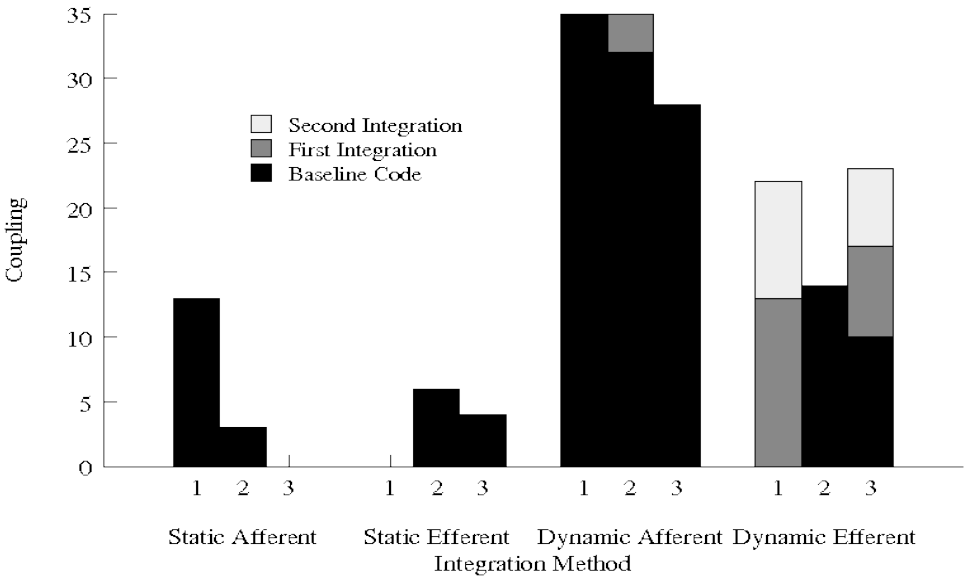

Figure 10.1: Summary of coupling changes for the various integration approaches to the Pragmatics task

10.2 Complexity Change Cost

In this section we present the summary metrics related to complexity changes due to integration. The summary values for LOC changes, McCabe complexity changes, and the resulting complexity change cost are shown in Figures 10.2, 10.3, and 10.4. Using SOA, we observe that significant complexity is involved in the integration. This complexity is due largely to the need for mapping messages from the GUI to messages for the Backend within the Pragmatics component and similarly for the updates coming from the Backend to the GUI. However, the complexity change cost remains relatively constant over the different integration stages due to the reuse of the same message mapping code for all subsequent message additions. This result is important since an ESB infrastructure component that is responsible for remapping messages based on content will have nearly identical characteristics to the Pragmatics component. The reuse of such an infrastructure component allows for the upfront expense in handling the complexity with subsequent reduced expense and increased value from hiding the complexity and using a component that has already been tested and does not change.

In the Player approach, we observe a large jump in complexity change cost due

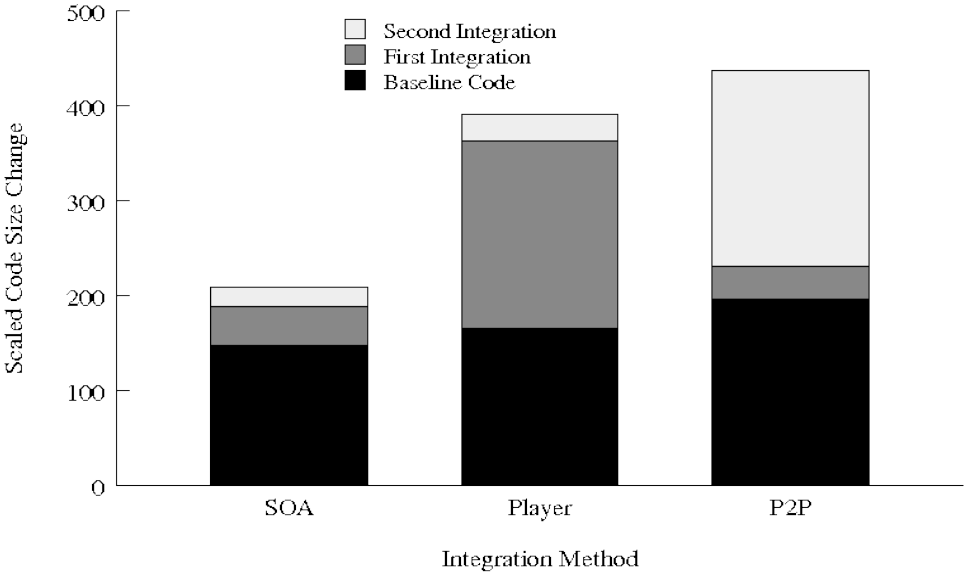

Figure 10.2: Code volume changes across the baseline, first, and second integration stages for the Pragmatics task

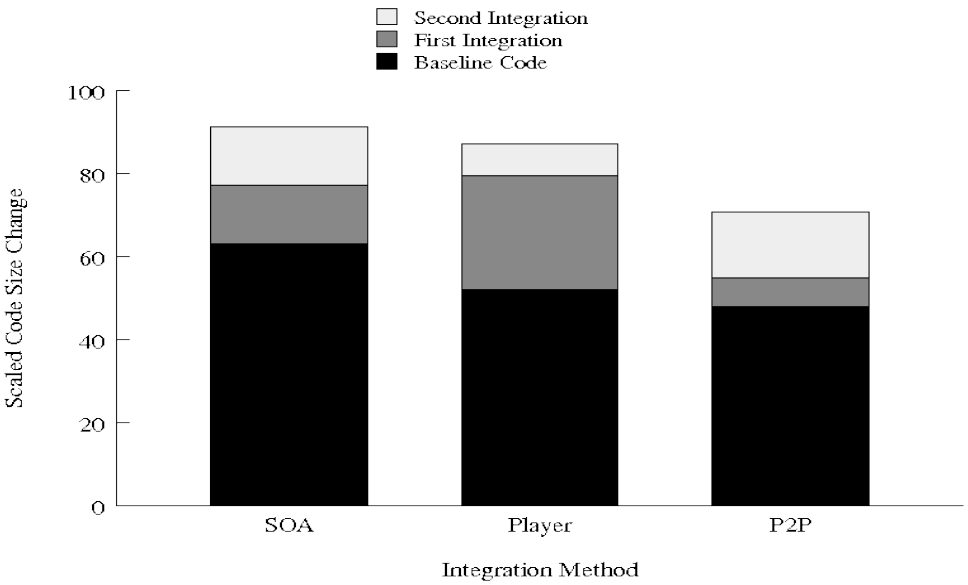

Figure 10.3: Complexity changes across the baseline, first, and second integration stages for the Pragmatics task

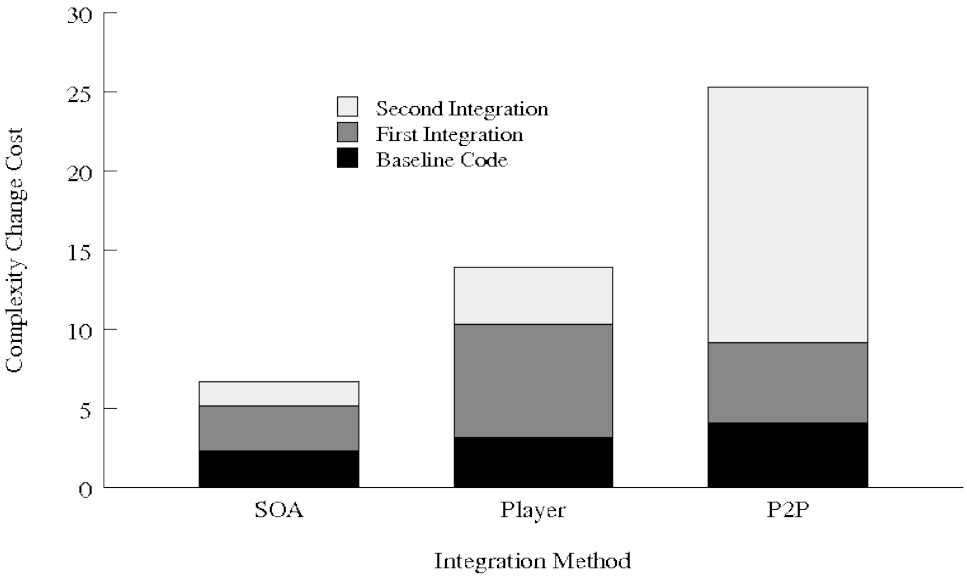

Figure 10.4: Complexity change cost across the baseline, first, and second integration stages for the Pragmatics task

to the introduction of the messages into the system at the first integration phase. In the subsequent integration, however,the values drop sharply and remain lower. As in the WDB integration task, this is due largely to the similarity of the message handling code for all subsequent messages added to the system.

For the P2P approach, the complexity change cost is high and reflects the fact that addition of multiple messages and using components that function as both client and server increases the difficulty of working with the source code. Complexity change cost for the P2P integration also increases significantly after the addition of the second message for the same reasons discussed in Chapter 8. This is because P2P approaches require creation of capabilities that exist in both the SOA and Player approaches.

Part V

Player Module Integration

11. PLAYER MODULE INTEGRATION APPROACH

As a final application of the SOA approach, we take an existing Player driver and integrate that driver into the SOA architecture. The goal is to make the functionality contained in an existing Player driver library accessible without the use of the Player server. Additionally, integration in this way means that the consumer application is not aware that the functionality is implemented in a Player library and does not require use of the Player libraries or configuration files.

11.1 Requirements Derived From Player

Many of the drivers provided with the Player distribution support multiple interfaces. Thus, making the driver accessible means that a client application can make calls on the supported interfaces and expect the driver to provide the functionality. Additionally, since multiple drivers may support the same interface, the client must specify the specific driver against which the interface call should be made. To illustrate the application of the SOA approach to integrating existing functionality, we focus on a particular interface of one driver and build up the integration in stages as was done in the previous integrations.

We selected the *position2d* interface as the Player interface integration target. This interface is supported by many of the drivers supplied in the Player distribution but no driver supports all of the interface methods defined for *position2d*. The particular driver we chose was the *RFLEX* driver. This driver supports the following interfaces and methods:

- position2d
  - PLAYER_POSITION2D_REQ_SET_ODOM

  - PLAYER_POSITION2D_REQ_RESET_ODOM
  - PLAYER_POSITION2D_REQ_MOTOR_POWER
  - PLAYER_POSITION2D_REQ_VELOCITY_MODE
  - PLAYER_POSITION2D_REQ_GET_GEOM
- sonar
  - PLAYER_SONAR_REQ_POWER
  - PLAYER_SONAR_REQ_GET_GEOM
- ir
  - PLAYER_IR_REQ_POWER
  - PLAYER_IR_REQ_POSE
- Bumper
  - PLAYER_BUMPER_REQ_GET_GEOM

The *SET_ODOM* and *RESET_ODOM* methods are the two methods supported in the first and second integration stages, respectively.

To integrate with an existing Player driver, without modification to the driver, requires that the service provider functionality that exposes the driver capabilities use the mechanisms that a Player server uses with drivers. This requirement means that we must initialize a driver using the Player configuration mechanisms, pass messages to the driver, and ensure that those messages are formatted according to the Player message format. These requirements are satisfied by linking in the Player shared libraries and calling the driver methods directly. Player drivers must implement a defined interface that allows the Player server to manage all drivers in the same way. The service provider wrapper makes use of that interface to ensure that the driver is initialized correctly and to handle messages.

Messages are passed from the Player server to the driver by placing inbound messages into a queue which is processed within the driver's implementation of *ProcessMessages*. The *ProcessMessages* method is one of the standard interface methods

that all Player drivers must implement. Within a driver, *ProcessMessages* is called from within an event loop initiated when the driver's *Main* method is called. Within *ProcessMessages*, messages in the queue are checked for the type and subtype of the message to determine if the message is handled by the driver. If not, the message is ignored or discarded. Otherwise, the message handler performs the functionality specified by the message type and subtype which identify the interface method being invoked. The implementation of the interface method is provided within the *ProcessMessages* body or may be invoked via a local, private method. Finally, all driver interface methods pass a single struct containing the parameters required by the method. The structs are defined in a driver definition file with several different structs defined for different methods.

11.2 Implementation Approach

As a result of the previous requirements, our integration approach was to create an API for consumer applications consisting of the methods *SetOdometry(double x, double y, double yaw)* and *ResetOdometry()*. These API calls mapped into IceStorm messages, *Provider_SetOdometry_Request* and *Provider_ResetOdometry_Request* for transmission on the ESB. On the provider side, the provider application subscribed to the IceStorm messages and upon receipt of one of the two messages, converted that message into the equivalent Player messages. The provider creates a message header according to the Player message header format, attaches the message payload, and inserts the message onto the driver's inbound message queue. During event loop processing in the driver, the message is removed from the inbound queue and processed within the *ProcessMessages* method.

To hide the Player-specific struct detail from service users, we defined the consumer side API with the individual parameters contained in the structs instead of struct equivalents. Then, to facilitate the producer side mapping into Player messages, we made Slice class definitions corresponding to the Player structs used by *position2d* and used the generated forms filled with the parameter data supplied by the API

call as message payloads to the provider. When the provider receives a message, the payload is transferred into the correct Player struct and incorporated into the Player message put on the inbound queue.

The consumer and provider were SOA-enabled by using the existing IceStorm protocol adapter developed for the World Database integration tasks and the ServiceUtilities component developed for the Pragmatics integration task. Additionally, the components mediating the translation between the different data representations, while created manually for this integration task, can be generated from the service descriptor file by including data type mapping sections.

In the next chapter we present the results of the measurements made on the software during each stage of the integration.

12. PLAYER MODULE INTEGRATION RESULTS

This chapter provides additional details of the SOA approach to integrating a legacy Player driver module. Additionally, we present the measurement results of LOC, McCabe complexity, and complexity change cost for the SOA approach. The data values are grouped by the integration phase in which they were collected and summarized by the location of the software element from which the value was obtained.

This integration task is different from the WDB and Pragmatics integration since we already have the Player integration and the complexity change cost for Player is zero. Additionally, for this task we were only interested in understanding how much effort is required to apply the SOA approach when no effort was required to use Player. Thus, we did not use the point-to-point approach and did not make coupling measurements.

Based on the description of the approach given in the Chapter 11, we created the following classes and applications:

1. consumer.cpp—standalone consumer test application
2. provider.cpp—standalone provider test wrapper application
3. player-p2d-mediator—base class for message translation between consumer API, ESB, and Player formats
4. player-p2d-mediator-consumer—class derived from player-position2d-mediator for message mediation between consumer API and ESB formats
5. player-p2d-mediator-provider—class derived from player-position2d-mediator for message mediation between ESB and Player formats

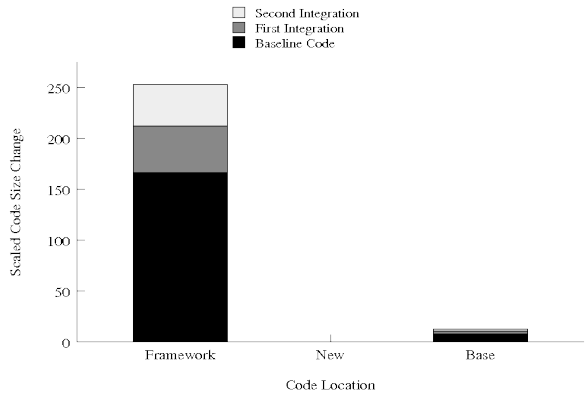

Figure 12.1: Code volume changes by location across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph can be found in Appendix I, Tables I.3, I.7, and I.11.

6. player-p2d-consumer—consumer side implementation class for the Slice interface definitions

7. player-p2d-provider—provider side implementation class for the Slice interface definitions

All measurements and metric calculations were made on these classes and applications.

12.1 Results Discussion

In this section we present the measurements and metrics obtained from the code resulting from the integration of the Player module. The results for code size change, code complexity change, and the complexity change cost categorized by location are given in Figures 12.1, 12.2, and 12.3. The summary results are shown in Figures 12.4, 12.5, and 12.6. In the next set of figures, the location data is summarized and contrasted against the Player values (all of which are zero for the purposes of this integration). In this integration task, all complexity change cost and effort for Player are zero. In the SOA approach, however, we observe a similar pattern as in previous integrations: the initial complexity change cost is large but decreases or stabilizes at a near constant value with subsequent integrations. The large complexities

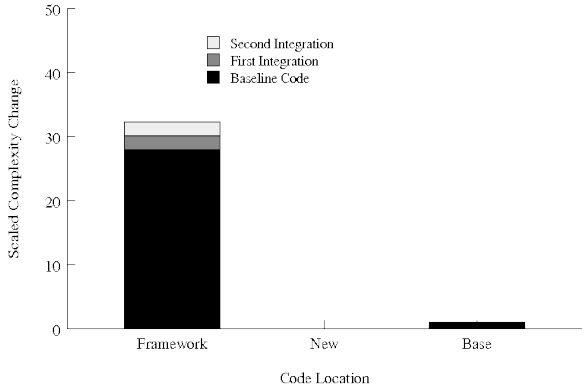

Figure 12.2: Code complexity changes by location across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph can be found in Appendix I, Tables I.4, I.7, and I.11.

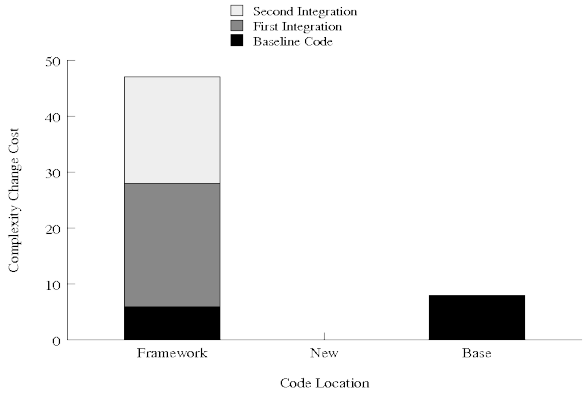

Figure 12.3: Complexity change cost by location across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph can be found in Appendix I, Tables I.4, I.7, and I.11.

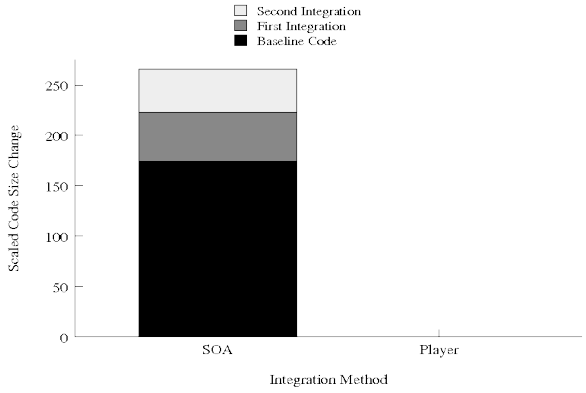

Figure 12.4: Code volume change across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph was derived from the tables in Appendix I.

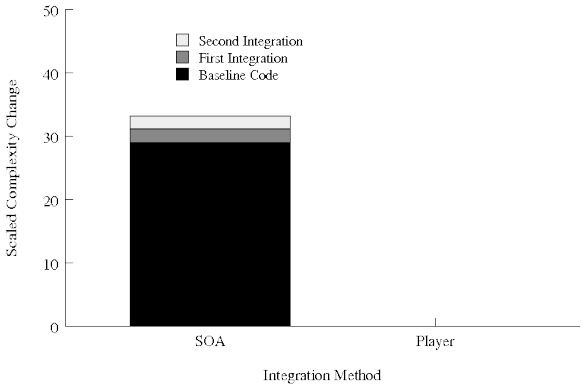

Figure 12.5: Code complexity change across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph was derived from the tables in Appendix I.

Figure 12.6: Complexity change cost across the baseline, first, and second integration stages for the Player module task. Detailed data for the graph was derived from the tables in Appendix I.

and code change values observed at each stage are due to the necessity of transforming parameter data at multiple locations in the service chain into a form usable by the integrated Player driver. These large values could potentially be reduced through the use of a mediation service with knowledge of specific data mappings or a data representation such as XML that permits transformations through declarative means such as the use of XML Schema Language Transformation (XSLT). These constructs represent more generalized mediation of data transformation and their use avoids the need to explicitly code data transformations in the provider or consumer call stack.

Part VI

Discussion and Conclusions

13. ANALYSIS AND INTERPRETATION

In the previous chapters we presented and summarized the details of the measurements and metrics derived from the staged integrations performed for both the World Database and the SUBTLE integration tasks. In this chapter we will provide an analysis and interpretation of those summary values by examining how coupling influences integration tasks and how well the complexity change cost metric captures the effort to perform integration tasks.

13.1 Integration Approaches

Using a service-oriented approach to integration forces a change in the way we provide access to capabilities. When applied to a robotic platform, the SOA approach provides flexibility and more ease of use in consumer access to and provider delivery of service capabilities. However, this flexibility comes at a cost to the provider of framework components supporting service orientation. That cost is increased up-front complexity but with a rapid decline in future increases of complexity through further integration. Additionally, ESB framework components are not "throw away" components but can be reused by many different consumers, providers, or systems. This reuse allows for amortization of the initial cost of creation of the components across a larger user set thus reducing the individual costs for integration.

Consideration of the metric data also supports the notion that while the ESB model is extremely useful in supporting service-orientation, the particular implementation of an ESB is not as important as the infrastructure capabilities provided by the ESB implementation. The implementation of an ESB approach using Player as the underlying technology shows similar complexity costs as using a commercial messaging system. However, these similarities must be tempered with the understanding

that the ESB implemented on Player is limited by a lack of available infrastructure services and for which additional complexity costs are incurred to implement and provide. Use of the Ice and IceStorm products provide many of the necessary infrastructure services out of the box.

While the P2P approach to integration is still in use, the metric results for the impact of using such an approach suggest that adoption of higher level abstractions such as those provided by service orientation and Player, can increase the ability of roboticists to more quickly add, remove, or modify capabilities as mission or research conditions require. If we consider the approaches to integration in a hierarchy of abstraction, SOA-based approaches are at a very high level of abstraction. Frameworks, such as Player, ROS, 4D/RCS, and others mentioned in this thesis, provide elements at a level of abstraction below service orientation, while the P2P approaches are at the lowest levels of abstraction. In fact, frameworks like ROS and 4D/RCS incorporate elements of service-orientation through the use of software component architectures, of which service-orientation is an example.

13.2 Coupling

In the integrations using both the SOA and Player approaches, we observe in the metric data that the changes in coupling are accompanied by larger complexity values, particularly where the coupling is dynamic. This tracking of complexity with coupling is evident in the WDB integration task for the Player approach and in the Pragmatics integration task for the SOA approach. However, in both the SOA and Player approaches, the dynamic coupling and changes to that coupling are confined to components classified as framework components. Relocating the complexity to stable framework components that are not required to be changed hides that complexity to users of the SOA and Player approaches. The advantage of the SOA approach over the Player approach is through the ESB concept in which we have abstracted how requests are made, transmitted, and handled. This abstraction is reflected in the simple interface available to service consumers and service providers. In the

WDB case, the client interface consists of a single method, *execute*, the details of which are hidden from the consumer. Additionally, this abstraction adds flexibility to how consumer requests are handled. Through the dynamic coupling mechanisms, protocol adapters such as the IceStormAdapter can be swapped out or augmented with other protocols and the change is invisible to the service consumer and service provider. Similarly, the components managing the message exchange patterns such as the ServiceUtilities or MURI_DataRep components in the WDB integration task allow for dynamic changes to the available messaging patterns or even a move towards streaming data patterns. Again, such changes are not exposed to the service consumer or provider. Player provides limited capabilities in this area but the availability of a rich set of device definitions can be used in conjunction with an ESB framework to provide access to those devices in a flexible way.

13.3 Complexity Change Cost

Our definition of complexity change cost provides insight into the level of effort required to use a particular integration approach. In the case of the SOA approach, the complexity change cost values reflect that SOA enables integration changes with relatively low effort on the part of the consumer or provider. However, those same metrics indicate that the cost of using SOA is in the build out of the ESB or service architecture capabilities. So the cost has not been eliminated but relocated into a centralized collection of capabilities. This shifting of the complexity cost into framework effort is justified if such a framework allows for future reuse of components and thus distribution of the creation costs. Such reuse might include use of existing capabilities on new platforms, use of new capabilities on existing platforms, rapid exchange of capabilities on existing platforms, or discovery of existing capabilities on existing platforms. Additionally, creation of an initial ESB or use of an existing technology to provide an ESB, such as was done here with ZeroC IceStorm^{TM} or Player, provides an architectural advantage to integration by supplying a centralized capability to perform system wide activities such as health monitoring,

performance monitoring, compliance checking and monitoring, security, logging, and fault handling. All of these capabilities are important in the deployed use of robotic platforms and if not addressed in a flexible way, may limit the utility of the platform. Use of the complexity change cost metric provides an estimate of the effort required to add these capabilities based on the forecast complexity of the added capability.

The comparison of the different integration approaches using the complexity change cost also provides a means of performing future comparisons between architectural approaches. Use of estimated LOC changes and complexity changes can be used to derive complexity change costs for candidate architectures and for making a selection from those candidates. We have used the McCabe complexity metric as the basis for our calculations but other code complexity metrics exist and are useful in computing the complexity change cost. Similarly, we used LOC as an estimate of the effort but other, better effort indicators exist, such as function points, and these can be used to compute a complexity change cost. The choice of code volume (LOC in our usage) and code complexity (McCabe complexity in our usage) should be made appropriate to the architectures under consideration but the characteristics of the complexity change cost based on pre and post integration activities are equivalent with respect to the trends and relative magnitudes (relative as measured between architectural approaches).

14. DISCUSSION

In this chapter we will discuss the value of the research presented in this thesis. In addition, we will examine some limitations of the research and use these limitations to motivate a discussion of further research that serve as extensions to this research.

14.1 Contributions

The motivation for conducting this research was to show the value of applying best practice software engineering through an integration approach based on service orientation. This approach is inherently focused on the integration of heterogeneous systems into an extended system of collaborating interaction. To show the value of a service oriented approach, we constructed a set of metrics designed to capture the changes in complexity of the system resulting from integration activities and to associate that complexity change to an economic consideration of the level of effort required to effect the integration. Several contributions result from this research.

The first contribution is the definition and prototype implementation of a service oriented approach to integrating software capabilities within a robotic platform. The use of a service oriented approach to integration in general is not new or novel, having been used in the context of business enterprise architectures for several years. However, the application to robotic platforms is a new domain of application requiring a change of view of the robot platform as an enterprise that differs from the current practice. This perspective also requires a reinterpretation of what an enterprise looks like and with this reinterpretation, the assurance that the underlying concepts of enterprise architecture and service orientation have been addressed. The recent emergence of systems such as ROS [Quiqley et al., 2009] are indications that the robotics community is starting to look for more loosely coupled solutions to

constructing robotic platforms and some form of service perspective underlies those solutions.

A further contribution resulting from the enterprise perspective is that we now have tools to move from consideration of a single robotic platform as an enterprise. We can use the same techniques and conceptual model to view a single robotic platform as a member of an enterprise, where now the enterprise consists of multiple robots collaborating on tasks. With this shift in scale, we can now consider that each robotic platform while maintaining an internal enterprise, is also a producer or consumer of services in the larger enterprise. Similarly, collaborating groups of robots can be viewed as an entity collaborating in a much larger enterprise consisting of multiple groups of collaboration nodes. Thus, we have achieved a form of scalability in which the same architectural principle and approach may be applied at multiple operational scales. The utility of this scalability is that increasingly, robots do not operate in isolation and in many applications, particularly military applications, multiple organizations control multiple groups of robots and must collaborate effectively across ownership domains (see, e.g., U.S. Department of Defense [2009]).

The second contribution of this research is the definition of a metric for assessing the level of effort required to perform integration tasks through the change cost metric. While portions of the definition of the metric are not new, the application of the scale factors to effort and complexity metrics to account for the overall impact of the change, the use of total lines of code touched during the integration, and the parameterization by architectural location of the contributing factors to the metrics are new. The results of the application of the metric to actual project activities indicates that the metric captures not just the immediate difficulty in doing an integration task, but the overall trend of repeated integrations using the same architectural approach. In the case of SOA, the large upfront cost of the approach was justified in the significantly reduced costs to perform subsequent integrations. Similarly, for the point-to-point approach, the metric indicates a trend of increasing costs to add additional capabilities onto an already costly integration base. The parameterization by architectural location of the factors contributing to the complexity change cost metric

also support the concept that moving or hiding complexity within general use framework components adds significant value to the users of service oriented approaches to integration. This complexity hiding removes the user from direct contact with the integration complexity and further spreads the cost of that complexity across potentially many similar users.

Finally, a third contribution lies in the definition of a service descriptor suitable for use within robotic applications. Use of such a service descriptor allows for the dynamic generation of service capabilities as needed and in a form suitable for use by consumers implemented in a variety of programming languages using a variety of platforms and a variety of access models. Typical SOA service descriptors are for services accessed over HTTP using SOAP packaging forms or for the provision and consumption of services using a Java implementation. This condition exists because the origin of services within the business community has been driven by the use of Web Services making use of predominantly web transport protocols and the fact that many businesses have implemented business logic within application servers such as the Java Enterprise Edition (formerly J2EE) platform or the Microsoft .NET platform. In addition to the use of the service descriptor for service generation, a portion of the metrics collection process is embodied within the code generation process based on the service descriptor. The dynamic coupling of the generated components is easily obtained from the generation process and since the metrics for complexity change cost do not include generated code in the computation, the measurements needed for the factors of the complexity change cost are isolated to a smaller number of easily measured modules.

14.2 Limitations

No research is performed without limitations to what was or what could be done. This research is no exception. Some of the limitations are due to time constraints while others are a result of boundaries imposed to prevent the principles being investigated from being obscured. In this section, we describe the most prominent

limitations.

Despite our description of this approach as service oriented, there some key features of service orientation that are not described in this research. These features include SOA governance, management, security, and service discovery. SOA governance is a process in which stakeholders mutually agree to the policies and guidelines for participation in the service oriented system. Management is the complementary concept to governance in its monitoring and enforcement of the governance policies and guidelines. We have not explicitly described these for the application of service oriented architectures on robotic platforms. However, there is implicit policy defined in the use of the service descriptor, the use of the ESB, and the use of the abstract communications base class. These policies are enforced through code generation and the monitoring capabilities provided by the ZeroC IceStorm^{TM} capability to track message flow. Similary, use of ZeroC IceStorm^{TM} provides an implicit form of service discovery which could have been encapsulated within a module for querying the message broker for details about the services available via the set of registered topics. Security in a SOA environment is of the utmost importance but since we were concerned with demonstration of the technical feasibility of the approach, we felt that the additional complexity of securing services would have obscured the more fundamental concept of technical feasibility.

Readers familiar with SOA may be asking where the Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP) descriptions are. We have purposely ignored the notion of "web services" since the intent is to show how service orientation can be applied to environments in which web protocols are not dominant or even present. Our ESB approach is based on simple TCP/IP protocols and can thus be easily extended to the HTTP protocols prevalent in use by web services if needed. Web services represent only a portion of service orientation and are based on ubiquitous web protocols. We are more concerned with the ability to support service orientation with protocols other than web protocols. While this is not truly a limitation, the association of SOA with web services causes much confusion when services are discussed that are not web based.

We have limited the granularity of how location assignments are made for components to just three categories: base, new, and framework. Finer grained decompositions are possible, particularly within the framework location, that provide insight into exactly where complexity is being accumulated and how this provides benefits for different architectural approaches. An example would be the decomposition of the framework location into localizations associated with the transport protocol, the message exchange patterns, infrastructure services, and basic messaging support. Since these areas are relevant to service orientation, a detailed analysis of how different architectural approaches handle these localizations would be useful when comparing those alternative approaches to SOA approaches.

The possibility exists that particular combinations of code volume metrics and complexity metrics used in computing the Complexity Change Cost might exhibit insensitivity to the scope of changes for particular integrations. However,the choices we used did not show any specific insensitivity to the types of changes we anticipated in performing integration. Further investigation would be needed to explore various combinations of code volume and code complexity to gain insight into the sensitivity of the metric to the particular choices of component metrics. We did not perform this analysis since our investigation was focused on the application of service orientation to the robotic platform environment.

Finally, we restricted our ESB to handling strictly messaging over TCP/IP. This restriction was made because of a lack of support for other transport protocols in performing basic messaging. In a true SOA, such a restriction would not need to be imposed or the governance policies would specify the conditions under which TCP/IP was applicable, when the use of a service interface was appropriate, and alternatives and the appropriate context for those alternatives to the use of TCP/IP messaging. As an example, consider how streaming video might be used in a service interface. Definition of a service interface to retrieve such a data stream by single frames would be inappropriate because of the large overhead and latency associated with making a single service call. Governance policies can be invoked in such cases to specify that the requirements for use of a streaming data source are to invoke a service

interface to gain access to and manage the streaming data source and then associate a point-to-point connection over which the actual data may be provided. This type of out of band control capability is similar to that used in CANBus applications where separate communication lines exist for the movement of control and data. We have provided a similar capability in the separation of the data stream as "Data Services" and the source of such data streams in "Sensor Services."

14.3 Extensions

The architecture and approach detailed in this thesis form a foundation for continued development. There are several directions for further developoment that were not pursued including those items described in the previous section on limitations. Those areas include the following:

1. Adaptation of the service infrastructure for real-time or near real-time capabilities. This adaptation includes
   (a) Use of real-time wire protocols such as CAN-bus and ProfiBus as service infrastructure transport protocols,
   (b) Service access under real-time and near real-time requirements, and
   (c) Performance analysis and management of real-time service use.
2. On demand, adaptive selection, management, and mediation of protocols by the service infrastructure based on service consumer specified Quality of Service (QoS) profiles
3. Extension to collaborative systems involving multiple heterogeneous autonomous platforms
4. Further investigation into the sensitivity of the complexity change cost metric including application to finer granularity localizations
5. Further build-out of the service infrastructure

The following paragraphs provide more detail for each of these items.

14.3.1 REAL-TIME SERVICE ACCESS

The prototype implementation of our service-oriented architecture relies on a message oriented middleware (MoM) capability, namely ZeroC's IceStorm product, to implement the different service invocation patterns. MoM products are typically built upon internet protocols such as TCP/IP, UDP, or HTTP and rely on either a message broker, a message queue, or some combination of the two to enable message exchange between entities. These characteristics make MoM-only ESB solutions undesirable for working with autonomous systems due to the large number of components requiring communication buses that handle real-time command, control, and data traffic. Examples of these buses include

1. The Controller Area Network Bus (CAN-bus) [CANBus, 1991] which is a pervasive standard in systems requiring fast, timely delivery of control and data messages. Automotive control systems involving an electronic control unit are a typical application for CAN-bus. CAN-bus message "frames" are designed to be very small with bit-level encoding of packet metadata and up to 8 bytes of data payload. CAN-bus physical implementations rely on distributed master controllers, message broadcasting, and message prioritization based on the content of the message identifier. Most physical installations using CAN-bus provide separate communication lines for control packets and data packets.

2. The Process Field Bus (Profibus) [Profibus, 2002] which is used in production and process automation applications. Profibus is typically used in 2 variants: Decentralized Peripherals (Profibus DP) for sensor and actuator operation through a central controller or a network of controllers and Process Automation (Profibus PA) for monitoring through process control systems. Profibus is most useful for high speed, time critical applications and for applications with complex communication requirements.

Use of service functionality accessible via these protocols requires creation of protocol adapters to incorporate the protocol into the ESB. Protocol bridging and mediation services will be required for use of these services by consumers not using the protocols.

Given the deadline-based delivery requirements of these types of protocols, it seems unlikely that a brokering component as used in MoM applications will be useful or desirable due to the added latency. Practical use of these protocols will also require flexible data rate matching between the consumer and provider since the provider data rates can be as high as 12 Mbps (in the case of Profibus) and a consumer not on the Profibus is likely operating at a lower data rate. This impedance matching is a form of protocol mediation and can be facilitated through various means such as the use of a data buffering capability as described in the OMG's Data Distribution Specification [OMG, 2007] or through sampling approaches.

14.3.2 DYNAMIC AND ADAPTIVE RUNTIME PROTOCOL SELECTION

An underlying reason for using an ESB within an SOA is to accommodate federation of heterogeneous networks within an enterprise. The SOA approach presented in this thesis provides an abstraction intended to hide the details of the network type and interaction protocol used for a particular service interaction. This decoupling is extended by having a delegate mediate the call between the protocol adapter and the service participant. However, the architecture does not support an ability to dynamically and adaptively select the transport protocol from a set of available options, i.e., the protocol adapter choice is static. This static coupling creates strong coupling of the service participant to the initial choice of adapter. To remove this strong coupling, the protocol and connection should not be chosen until the request is made and the choice should be one that satisfies criteria associated with the particular invocation at both the consumer and provider endpoints. Delaying selection of the protocol and connection allows the service infrastructure to make the best selection based on knowledge of

1. The endpoint requirements,

2. The invocation type,

3. The current performance of known protocols,

4. With the addition of predictive capabilities, a knowledge of anticipated loading on specific protocols that have a higher priority, e.g., anticipation of control actions requiring the use of a specific channel for sending priority control signals

In practice, we might choose to make a predetermined choice for this protocol choice but in an operational environment with resource constraints, a better use of available resources might be to allow for other uses of such a channel but in a mode that allows for immediate pre-emption.

A candidate set of criteria to start from are QoS requirements specified by the service consumer. The QoS requirements are typically metrics on performance, capacity, and load tolerance that can be used to indicate a consumer's invocation needs. Similarly, service providers typically advertise QoS capabilities as a part of service registration or in a provider service descriptor. For example, choosing a CAN-bus adapter for an invocation that must send large data structures is not likely to be an optimal choice. However, if the consumer also has real-time data requirements, the message-oriented adapters are not suitable unless the only available services use message-oriented channels. In addition to the specific QoS parameters, the consumer can specify a prioritization of the requirement parameters to assist in selection decisions that may require a trade-off choice. The selection of an adapter is then made based on the parameter values at invocation time, the available channels, and an understanding of the current performance of those channels. In support of this capability, the service infrastructure or the ESB must provide a mechanism for monitoring and persisting the relevant QoS metric data for known adapters and channels as well as maintaining up-to-date performance information on end-to-end connections using those adapters. A separate mechanism must be provided by the adapters for reporting relevant QoS data and to support dynamic loading of the protocol adapters into the service participant execution space.

Using this type of approach for protocol selection views protocol choice as a service request made to the service infrastructure. Thus, any service request requiring a protocol selection becomes an orchestrated service in which the infrastructure coordinates the marshalling of the required resources and then directs the execution

sequence of those resources. This view of protocols in the service-oriented environment is new. Commercial ESB products tend to predetermine the choices of protocols and channels allowing some flexibility by defining multiple protocol choices for a particular service. However, the flexibility only extends to use of the protocols in the stated order if a protocol higher in the sequence is unavailable. This predetermination requires analysis and foresight by the architect or designer and is of limited value in environments for which protocols may be routinely added or removed.

14.3.3 MULTI-AGENT EXTENSIONS

The demonstration of the prototype architecture was done through a limited set of integration activities confined to a single robotic platform. There is nothing in any of the architectural discussions that requires the consideration of a single platform: we made the choice as a way to constrain the problem. Extending the service oriented concepts beyond the individual platform means that we can consider the individual platform as a potential service or as a potential service provider. The enterprise is then extended to subsume groups of robots where now the service package concept is focused on the types of services provided by a particular group of robots. Similarly, groups of collaborating robots can be viewed as potential service entities participating in an enterprise in which there exists multiple similar groupings of robots, other types of platforms, and humans. Use of this view allows for the creation of *ad hoc* collaborations when needed without significant development effort. The approach also allows for the augmentation of existing capabilities through the addition of platforms providing needed capabilities such that the existing platforms do not require modification.

14.3.4 COMPLEXITY CHANGE COST METRIC INVESTIGATION

Our construction of the complexity change cost metric was based on the assumption that code volume metrics as an indicator of level of effort allows an assignment of a unit "cost" to the complexity changes resulting from integration activities. The initial choice of SLOC as a level of effort was made since many of the tools

available provide SLOC measurements and many software organizations continue to use SLOC as an effort estimator. Implicit in the initial assumption is that level of effort and code complexity are correlated. We have not proven that in this thesis but instead used the assumption as the basis for an operational approach to a rudimentary validation of the metric through the measurement of actual integration tasks. The results obtained for the complexity change cost metric fit with our experience in performing integrations but this is insufficient evidence on which to accept the validity of the metric. A further investigation into how the metric tracks complexity and effort when different code volume and complexity metrics are used. This analysis is particularly important when we consider that SLOC measurements are problematic in service environments where a considerable portion of code is either unavailable or automatically generated. Automatic generation of code artificially inflates code volume metrics and such metrics are being deprecated for use in these environments. In addition, an analysis of the sensitivity and limits to the validity of the metric require an in-depth analysis. Finally, our parameterization of the metric by identification of the location of the changes requires further investigation to understand how well the metric captures the effort cost when a finer grained localization is used.

14.3.5 EXPANSION OF SERVICE INFRASTRUCTURE

The approach we used to provide an ESB was simple in that we ignored larger service infrastructure capabilities such as security, logging, auditing, protocol mediation, management, discovery, and governance. These additional capabilities are necessary for a fully capable service oriented architecture. Having demonstrated that a service oriented approach provides utility to integration tasks, the next step is to build-out the service infrastructure in conjunction with the expansion of the protocol support to realtime protocols. This more complete ESB is the ultimate goal and will require investigation of where existing commercial technologies are applicable and where components must be constructed to fill the gaps. An additional optional task is to consider how a standard service API for the various service categories could be created. Use of a standard API means that the service infrastructure specifies

an API appropriate for each service type and enforces its use. For example, a Data Service API would include Create, Retrieve, Update, and Delete (CRUD) operations commonly associated with database operations but would extend those operations to non-database data sources such as sensor data streams. We are continuing effort in these areas but because they are not fully developed, have not included discussion of them except in this section.

15. CONCLUSION

In this thesis we have presented an approach to integration for robotic platforms that is based on best practice software engineering principles, sound software architectural principles, and a change in the perspective of what the robotic platform represents. Instead of the view of the robot as a collection of hardware and software supporting that hardware, we have shown that viewing the robot as an enterprise system in which services are offered and consumed provides a basis for the use of enterprise architecture approaches to handling integration. Through the completion of project based integration tasks in support of the SUBTLE MURI project, we were able to demonstrate that use of a service oriented approach facilitates integration naturally and allows for easier addition of new capabilities as needed. Additionally, we defined a metric, the complexity change cost, to be used as a measure of the effort required to change the complexity of a system. This metric allowed us to compare integration approaches based on different architectural principles in a way that captured the utility of each architecture in supporting integration.

Service orientation is fundamentally about incorporating heterogeneity into systems without requiring significant recoding, rework, or new component creation. Robotic platforms are models of heterogeneity with the variety of sensors, software packages, and applications in use on a single platform at any instant. Extension of this concept beyond the individual platform provides a model for collaborative uses of robots where the integration between the robots is done using the same architectural approach as the integration on the individual robot. The work described in this thesis is a foundation for further extensions to the service concept and the possibilities enabled by service orientation will permit new ways of thinking about how robots are to be used.

Appendices

A. SERVICE DESCRIPTOR

The code generator described in Section 5.2.1.3 uses an XML description of the service and the characteristics of either the consumer or provider to create the software components needed to expose or use the service. The XML description is an instance document of an XML Schema Definition (XSD) for services called the Service Descriptor. Separate definitions of the service profiles for the consumer and provider must be made. The definitions in the service descriptor XSD are presented in the next section accompanied by a brief explanation. Taken *in toto*, these definitions form the service descriptor.

A.1 Service Descriptor Definitions

The XML target namespace for the definitions in the service descriptor schema is "tns" and is defined in the XSD as

Service Descriptor Namespace

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.w3c.org/services"
        xmlns:tns="http://www.w3c.org/services"
        elementFormDefault="qualified">
```

The definitions in the remainder of this section reference that target namespace.

A.1.1 SERVICE CALL PARAMETER

A service call parameter type is an argument defined by the service provider in the service interface specification. The default value is not required but is recommended since in some cases the code generator may not be able to determine a reasonable default.

Service Call Parameter

```
<complexType name="parameterType">
  <sequence minOccurs="1">
    <element name="parameterName"
             type="string"
             maxOccurs="1"/>
    <element name="parameterDataType"
             type="tns:dataType"
             maxOccurs="1"/>
    <element name="defaultValue"
             type="string"
             minOccurs="0" maxOccurs="1"/>
  </sequence>
</complexType>
```

A.1.2 TARGET LANGUAGE

The target language is the programming language output from the code generation process. The prototype implementation supports code generation in Java, C++, and Python.

Target Language

```
<simpleType name="targetType">
  <restriction base="string">
    <enumeration value="java"/>
    <enumeration value="c++"/>
    <enumeration value="python"/>
  </restriction>
</simpleType>
```

A.1.3 MESSAGE EXCHANGE PATTERN

The message exchange pattern (MEP) represents the protocol for the types of messages exchanged between the consumer and the provider. We have identified 5 MEPs based on those defined in Josuttis [2007], but in the prototype implementation, only the Request/Response (QueryResponse in our usage) and One Way or AsynchronousRequest patterns are implemented. The remaining 3 MEPs are variants of the QueryResponse and AsynchronousRequest. The point-to-point MEPs, SynchronousP2P and AsynchronousP2P, demand a tighter coupling of the consumer

to the provider but are defined for those cases where the round trip execution time would limit the utility of the service.

Message Exchange Pattern

```
<simpleType name="communicationType">
  <restriction base="string">
    <enumeration value="QueryResponse"/>
    <enumeration value="AsynchronousRequest"/>
    <enumeration value="SynchronousP2P"/>
    <enumeration value="AsynchronousP2P"/>
    <enumeration value="LightweightEvent"/>
  </restriction>
</simpleType>
```

A.1.4 PACKAGING FORM FACTOR

The packaging form factor type is used to indicate to the code generator how to package the generated adapter and proxy libraries. This type is used in conjunction with the Operating System type to create the appropriate binary format and select the necessary set of compile and link flags for the generated makefile.

Packaging Form Factor

```
<simpleType name="operationType">
  <restriction base="string">
    <enumeration value="executable"/>
    <enumeration value="shared-lib"/>
    <enumeration value="static-lib"/>
    <enumeration value="object-file"/>
  </restriction>
</simpleType>
```

A.1.5 OPERATING SYSTEM

The operating system type is provided to ensure the generated code and accompanying build files are correct for the platform. Note that in the current data model, specification of a different hosting platform requires definition of a new profile. The prototype implementation supports Linux fully and only BSD formats for use on Mac OSX.

Operating System

```
<simpleType name="hostType">
  <restriction base="string">
    <enumeration value="Windows"/>
    <enumeration value="Linux"/>
    <enumeration value="MacOS"/>
  </restriction>
</simpleType>
```

A.1.6 SERVICE DATA TYPES

The Service Data Types are defined for use as service call parameter and return types. The type names are a subset of the data types defined in the ZeroC SLICE language but are sufficiently generic to support a large percentage of uses. The "blob" type is a binary large object and is to be considered as a collection of binary data with an undefined or unknown structure. This type is provided to support imagery, audio, and other high volume binary data.

Service Data Types

```
<simpleType name="dataTypes">
  <restriction base="string">
    <enumeration value="void"/>
    <enumeration value="int"/>
    <enumeration value="float"/>
    <enumeration value="string"/>
    <enumeration value="blob"/>
  </restriction>
</simpleType>
```

A.1.7 SERVICE IMPLEMENTATION FORM FACTOR

The service implementation form factor type is used to describe how the service provider has implemented and packaged the service functionality. This type is only relevant to the "provider" role and is used to create the mechanisms needed for invoking the service functionality when a request is received at the service provider. Note that all 3 elements are required and intended to be interpreted as a fully qualified path to the executable implementation of the service functionality. For languages

with packaging constructs such as Java packages or C++ namespaces, the "impl-class" entry should be the fully qualified class name. The "bin-location" element is the file system directory path to the binary file containing the executable code. This path is assumed to be relative to the location of the adapter and proxy libraries or executables.

Service Implementation Form Factor

```
<complexType name="implType">
  <sequence minOccurs="1">
    <element name="impl-method" type="string"/>
    <element name="impl-class" type="string"/>
    <element name="bin-location" type="string"/>
  </sequence>
</complexType>
```

A.1.8 CODE GENERATION PROFILE

The code generation profile type is used to identify which artifacts to generate and what platform constructs should exist in those artifacts.

Code Generation Profile

```
<complexType name="serviceGenerationProfile">
  <sequence minOccurs="1">
    <element name="target-language" type="tns:targetType"/>
    <element name="operation-model" type="tns:operationType"/>
    <element name="host-model" type="tns:hostType"/>
  </sequence>
</complexType>
```

A.1.9 SERVICE ACTOR

The service actor type describes whether the code is to be generated for the service provider or the service consumer.

Service Actor

```
<complexType name="serviceActorType">
  <choice>
    <element name="provider-profile"
             type="tns:serviceGenerationProfile"
             minOccurs="0" maxOccurs="1"/>
    <element name="consumer-profile"
             type="tns:serviceGenerationProfile"
             minOccurs="0" maxOccurs="1"/>
  </choice>
</complexType>
```

A.1.10 SERVICE VERSION NUMBER

The service version number is provided to allow for consistency and compliance checking. The prototype implementation does not look at service versions to ensure consistency between the consumer and provider but a robust SOA would provide version checking at many different levels, e.g., data type, service, or service level agreement versioning, to facilitate enterprise management functions.

Service Version Number

```
<simpleType name="versionType">
  <restriction base="string">
    <pattern value="([0-9])+.([0-9])+.([0-9])+([a-z])*"/>
  </restriction>
</simpleType>
```

A.1.11 SERVICE DEFINITION

The service definition type is the service profile for a single service.

Service Definition

```
<complexType name="serviceType">
  <sequence>
    <element name="name"
             type="string"
             minOccurs="1" maxOccurs="1"/>
    <element name="version"
             type="tns:versionType"
             minOccurs="1" maxOccurs="1"/>
    <element name="implementor"
             type="tns:implType"
             minOccurs="1" maxOccurs="1"/>
    <element name="service-actor"
             type="tns:serviceActorType"
             minOccurs="1" maxOccurs="1"/>
    <element name="comm-model"
             type="tns:communicationType"
             minOccurs="1" maxOccurs="1"/>
    <element name="return"
             type="tns:dataTypes"
             minOccurs="1" maxOccurs="1"/>
    <element name="parameters">
      <complexType>
        <sequence minOccurs="0" maxOccurs="unbounded">
          <element name="parameter"
                   type="tns:parameterType"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

A.1.12 SERVICE PACKAGE NAMING

The service package naming type is used to provide namespace separation for different services. The namespace is similar to Java package naming.

Service Package Naming

```
<complexType name="servicePackageNameType">
  <sequence>
    <element name="packageName"
             type="string"
             minOccurs="1" maxOccurs="unbounded"/>
  </sequence>
</complexType>
```

A.1.13 SERVICE PACKAGE

The service package type is provided to allow for grouping of services having some conceptual or logical relationship and for which the service profile definitions would exist in the same package structure. The type is recursively defined to allow for the definition of service profiles at any level of a package naming hierarchy. With respect to code generation, the service package structure is used to create a file system directory structure containing the generated code and build files.

Service Package

```
  <element name="service-package">
     <complexType>
       <sequence>
         <element name="servicePackageName"
                  type="tns:servicePackageNameType"
                  minOccurs="1" maxOccurs="unbounded"/>
         <element name="service"
                  type="tns:serviceType"
                  minOccurs="1" maxOccurs="unbounded"/>
         <element ref="tns:service-package"
                  minOccurs="0" maxOccurs="unbounded"/>
       </sequence>
     </complexType>
  </element>
</schema>
```

B. QUALITY ATTRIBUTE DESCRIPTIONS

The quality attributes referred to in Section 5.1.1 are defined and explained in detail in this appendix. For more information on quality attributes, refer to Barbacci et al. [1995].

Extensibility The quality of a system that allows the system capabilities to be extended by the addition of new functionality when needed. Extensibility in an architecture can be obtained through mechanisms designed to support the addition of new components without the need to rebuild the system. This type of architecture typically supports the integration of new components through changes in configuration information. Examples of extensible systems are systems embodying messaging systems, plug-and-play component technology, or object location services[1] such as the Naming Service in a CORBA system. Other types of extensible systems provide "hooks" from which a developer can define new functionality and attach it to the existing system. Examples of these types of systems include Emacs and the Eclipse IDE.

Maintainability The quality of a system that supports normal system maintenance activities such as upgrades, patching, and configuration but also supports repair activities such as restoration of the system to full functionality within specified time constraints following an outage or fault. Maintainability can also be expressed as the ease with which maintenance teams can identify, isolate, debug, and repair system defects. Maintainability can be achieved through well planned logging and auditing capabilities, helpful error messages, and usable, up-to-date documentation.

[1] Object location services are software components that hold pointers to objects in a system and may be queried by other applications who wish to use these components.

Modifiability The quality of a system that permits changing parts of the system without impacting the structure or the operation of the entire system.[2]. Note that modifiability and extensibility are remarkably similar. They differ in that modifiability focuses on modification of the existing functionality to gain additional or new functionality whereas extensibility focuses on the addition of new functionality without modifying the existing functionality. Modifiability in a system may be achieved by limiting the interconnections between components used in the system. This limitation permits modification of a component without the effect spreading to other components even when those components have some connection to the modified component.

Performance The quality of a system that permits tasks to be performed in a minimal amount of time or within a predefined interval or in such a way that computational resource usage is minimized. Performance attributes can be obtained in architectures by minimizing the layers of code any request must traverse before being satisfied or by decreasing the granularity of data passed to and from procedure calls.[3] Performance may also be gained by addition of higher capacity hardware and implementation of certain algorithms (such as graphical or signal processing algorithms) directly in hardware. Further performance enhancement may be had by redundant hardware to allow concurrent request processing or through the addition of load-balancing mechanisms.

Reliability The quality of a system that permits the system to perform the same tasks or multiple tasks repeatedly without failure or requiring significant maintenance between tasks. We distinguish here between hardware reliability and software reliability although both are typically quoted as Mean Time Between

[2]Easily modified is a highly subjective term which is strongly dependent upon the system use and so will be defined differently for different systems.

[3]In Enterprise Systems, this data granularity severely affects network performance by increasing the amount of packet traffic in the network. For example, if we need a mailing address with a name from a database, we might have to use a couple of queries to get all of the information. In a networked system, this translates into a large number of data packets plus the additional query traffic. By structuring the data differently, we can ask for the complete dataset in a single query and reduce the network overhead. Similar concepts apply to procedure call granularities.

Failures (MTBF) and Mean Time To Return/Repair (MTTR) which are hardware reliability concepts. Reliability in a system may be achieved in many different ways, although it is often obtained through redundant hardware or software components. Other means of achieving reliability are to use transactional operations, apply quality of service characteristics to messaging, or to provide system health monitors for critical components.

Scalability The quality of a system that permits the system components to be used in small-scale systems as readily as in large-scale systems. This attribute implies that the component(s) may be used more or less intact on systems of any size and that the choice of components may be easily made based on the platform size, with a smaller subset of the components being used without degraded capability in a smaller system.[4]. Scalability may be achieved through the use of a minimal set of components applicable at all relevant scales. Scalability is also defined as the ability of the system to perform without degradation, maintain message throughput, or maintain round-trip processing times under increasing system loading.

Usability The quality of a system that permits an inexperienced user or developer to make use of the architecture or system with minimal training or skill. While the phrase "minimal training or skill" is very subjective, we mean it in the sense that the average user or developer will be capable of manipulating the system without requiring years of training. Another view of usability is through consideration of the learning curve required to be proficient with the architecture or system.

[4]Note that we intend degraded capability to imply that the system performs the available functions as easily in the small system as in the large system. The smaller system may be viewed as degraded, however, in the sense that fewer components may be used in a smaller system than in a larger system.

C. SOA WDB INTEGRATION DETAIL DATA

The coupling data obtained from the SOA baseline code are shown in Table C.1. The measures of LOC and McCabe complexity and the values for complexity change cost for the SOA baseline code are shown in Tables C.2 and C.3. Summarized coupling, LOC, and complexity metrics by location are shown in Tables C.4, C.5 and C.6.

C.1 First Integration

The coupling data obtained from the SOA first integration code are shown in Table C.7. The measured values of LOC, McCabe complexity, and complexity change cost for the SOA first integration code are shown in Tables C.8 and C.9. Summarized data by location are shown in Tables C.10, C.11 and C.12.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
ATRV_WDB_Impl_C	F	1	0	2	0
ATRV_WDB_Impl_S	F	1	0	2	0
ServiceUtilities	F	4	0	3	1
MURI_DataRep	F	1	0	1	0
IceStormAdapter	F	1	0	10	1
DataServiceMain	N	0	0	1	0
CPPTestClient	B	0	0	1	0

Table C.1: Detailed coupling data for the baseline SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
ATRV_WDB_Impl_C	F	0	22	22	0	0	1.0	22.0
ATRV_WDB_Impl_S	F	0	20	20	0	0	1.0	20.0
ServiceUtilities	F	0	87	87	0	0	1.0	87.0
MURI_DataRep	F	0	22	22	0	0	1.0	22.0
IceStormAdapter	F	0	92	92	0	0	1.0	92.0
DataServiceMain	N	0	13	13	0	0	1.0	13.0
CPPTestClient	B	0	14	14	0	0	1.0	14.0

Table C.2: Detailed LOC data for the baseline SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
ATRV_WDB_Impl_C	F	0	4	1.0	4.0	5.50
ATRV_WDB_Impl_S	F	0	4	1.0	4.0	5.00
ServiceUtilities	F	0	20	1.0	20.0	4.35
MURI_DataRep	F	0	4	1.0	4.0	5.50
IceStormAdapter	F	0	14	1.0	14.0	6.57
DataServiceMain	N	0	1	1.0	1.0	13.00
CPPTestClient	B	0	2	1.0	2.0	7.00

Table C.3: Detailed complexity data for the baseline SOA approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	8	0	18	2
Base	0	0	1	0
New	0	0	1	0

Table C.4: Summarized coupling data by location for the baseline SOA approach to the WDB integration.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	243	243	0	0	1.0	243.0
Base	0	13	13	0	0	1.0	13.0
New	0	14	14	0	0	1.0	14.0

Table C.5: Summarized LOC data by location for the baseline SOA approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	46	1.0	46.0	5.28
Base	0	1	1.0	1.0	13.00
New	0	2	1.0	2.0	7.00

Table C.6: Summarized complexity data by location for the baseline SOA approach to the WDB integration.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
ATRV_WDB_Impl_C	F	1	0	2	0
ATRV_WDB_Impl_S	F	1	0	2	0
ServiceUtilities	F	4	0	3	1
MURI_DataRep	F	1	0	1	0
IceStormAdapter	F	1	0	10	1
DataServiceMain	N	0	0	1	0
CPPTestClient	B	0	0	1	0

Table C.7: Detailed coupling data for the first integration in the SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
ATRV_WDB_Impl_C	F	22	22	0	0	0	1.00	0.00
ATRV_WDB_Impl_S	F	20	20	0	0	0	1.00	0.00
ServiceUtilities	F	87	87	0	0	0	1.00	0.00
MURI_DataRep	F	22	22	0	0	0	1.00	0.00
IceStormAdapter	F	92	92	0	0	0	1.00	0.00
DataServiceMain	N	13	14	1	0	0	1.08	1.08
CPPTestClient	B	14	18	4	0	0	1.29	5.14

Table C.8: Detailed LOC measurement data for the first integration in the SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
ATRV_WDB_Impl_C	F	4	4	1.0	0.00	0.00
ATRV_WDB_Impl_S	F	4	4	1.0	0.00	0.00
ServiceUtilities	F	20	20	1.0	0.00	0.00
MURI_DataRep	F	4	4	1.0	0.00	0.00
IceStormAdapter	F	14	14	1.0	0.00	0.00
DataServiceMain	N	1	1	1.0	0.00	0.00
CPPTestClient	B	2	2	1.0	0.00	0.00

Table C.9: Detailed complexity data for the first integration in the SOA approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	8	0	18	2
Base	0	0	1	0
New	0	0	1	0

Table C.10: Summarized coupling data by location for the first integration in the SOA approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	243	243	0	0	0	0.00	0.00
Base	13	14	1	0	0	1.08	1.08
New	14	18	4	0	0	1.29	5.14

Table C.11: Summarized LOC data by location for the first integration in the SOA approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	46	46	1.0	0.0	0.0
Base	1	1	1.0	0.0	0.0
New	2	2	1.0	0.0	0.0

Table C.12: Summarized complexity data by location for the first integration in the SOA approach to the WDB integration task.

C.2 Second Integration

The coupling data obtained from the SOA second integration code are shown in Table C.13. The measures of LOC, McCabe complexity, and complexity change cost for the SOA second integration code are shown in Tables C.14 and C.15. Summarized coupling, LOC, and complexity data by location are shown in Tables C.16, C.17 and C.18.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
ATRV_WDB_Impl_C	F	1	0	2	0
ATRV_WDB_Impl_S	F	1	0	2	0
ServiceUtilities	F	4	0	3	1
MURI_DataRep	F	1	0	1	0
IceStormAdapter	F	1	0	10	1
DataServiceMain	N	0	0	1	0
CPPTestClient	B	0	0	1	0

Table C.13: Detailed coupling data for the second integration in the SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
ATRV_WDB_Impl_C	F	22	22	0	0	0	1.00	0.0
ATRV_WDB_Impl_S	F	20	20	0	0	0	1.00	0.0
ServiceUtilities	F	87	87	0	0	0	1.00	0.0
MURI_DataRep	F	22	22	0	0	0	1.00	0.0
IceStormAdapter	F	92	92	0	0	0	1.00	0.0
DataServiceMain	N	14	15	1	0	0	1.07	0.0
CPPTestClient	B	18	22	4	0	0	1.22	0.0

Table C.14: Detailed LOC data for the second integration in the SOA approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
ATRV_WDB_Impl_C	F	4	4	1.0	0.0	0.0
ATRV_WDB_Impl_S	F	4	4	1.0	0.0	0.0
ServiceUtilities	F	20	20	1.0	0.0	0.0
MURI_DataRep	F	4	4	1.0	0.0	0.0
IceStormAdapter	F	14	14	1.0	0.0	0.0
DataServiceMain	N	1	1	1.0	0.0	0.0
CPPTestClient	B	2	2	1.0	0.0	0.0

Table C.15: Detailed complexity data for the second integration in the SOA approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	8	0	18	2
Base	0	0	1	0
New	0	0	1	0

Table C.16: Summarized coupling data by location for the second integration in the SOA approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	243	243	0	0	0	1.0	0.00
Base	14	15	1	0	0	1.0	1.07
New	18	22	4	0	0	1.0	4.89

Table C.17: Summarized LOC data by location for the second integration in the SOA approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	46	46	1.0	0.0	0.0
Base	1	1	1.0	0.0	0.0
New	2	2	1.0	0.0	0.0

Table C.18: Summarized complexity data by location for the second integration in the SOA approach to the WDB integration task.

D. PLAYER WDB INTEGRATION DETAILED DATA

D.1 Baseline

The coupling data obtained from the Player baseline code are shown in Table D.1. The measures of LOC, McCabe complexity, and complexity change cost for the Player baseline code are shown in Tables D.2 and D.3. Summarized coupling, LOC, and complexity data by location are shown in Tables D.4, D.5 and D.6.

D.2 First Integration

The coupling data obtained from the Player first integration code are shown in Table D.7. The measures of LOC,McCabe complexity, and complexity change cost for the Player first integration code are shown in Tables D.8 and D.9. Summarized coupling, LOC, and complexity data by location are shown in Tables D.10, D.11 and D.12.

D.3 Second Integration

The coupling data obtained from the Player second integration code are shown in Table D.13. The measures of LOC, McCabe complexity, and complexity change

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb_client	B	0	0	3	2
wdbinterf_client	F	1	1	3	0
wdbinterf_driver	F	0	1	5	0

Table D.1: Detailed coupling data for the baseline Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb_client	B	0	23	23	0	0	1.00	23.0
wdbinterf_client	F	0	82	82	0	0	1.00	82.0
wdbinterf_driver	F	0	68	68	0	0	1.00	68.0
F-Framework, B-Base, N-New								

Table D.2: Detailed LOC data for the baseline Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb_client	B	0	4	1.0	4.0	5.75
wdbinterf_client	F	0	16	1.0	16.0	5.13
wdbinterf_driver	F	0	10	1.0	10.0	6.80

Table D.3: Detailed complexity data for the baseline Player approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	1	2	8	0
Base	0	0	3	2
New	0	0	0	0

Table D.4: Summarized coupling data by location for the baseline Player approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	150	150	0	0	1.0	150.0
Base	0	23	23	0	0	1.0	23.0
New	0	0	0	0	0	1.0	0.0

Table D.5: Summarized LOC data by location for the baseline Player approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	26	1.0	26.0	5.77
Base	0	4	1.0	4.0	5.75
New	0	0	1.0	0.0	0.00

Table D.6: Summarized complexity data by location for the baseline Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb_client	B	0	0	3	2
wdbinterf_client	F	1	1	3	1
wdbinterf_driver	F	0	1	6	1

Table D.7: Detailed coupling data for the first integration in the Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb_client	B	23	28	5	0	0	1.22	6.09
wdbinterf_client	F	82	104	22	0	0	1.27	27.90
wdbinterf_driver	F	68	76	8	0	0	1.12	8.94

Table D.8: Detailed LOC data for the first integration Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb_client	B	4	5	1.25	1.25	4.87
wdbinterf_client	F	16	19	1.19	3.56	7.83
wdbinterf_driver	F	10	11	1.10	1.10	8.13

Table D.9: Detailed complexity data for the first integration Player approach to the WDB integration.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	1	2	9	2
Base	0	0	3	2
New	0	0	0	0

Table D.10: Summarized coupling data by location for the first integration Player approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	150	180	30	0	0	1.38	41.40
Base	23	28	5	0	0	1.22	6.09
New	0	0	0	0	0	1.00	0.00

Table D.11: Summarized LOC data by location for the first integration Player approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	26	30	1.15	4.60	9.00
Base	4	5	1.25	1.25	4.87
New	0	0	1.00	0.00	0.00

Table D.12: Summarized complexity data by location for the first integration Player approach to the WDB integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb_client	B	0	0	3	2
wdbinterf_client	F	1	1	3	2
wdbinterf_driver	F	0	1	7	2

Table D.13: Detailed coupling data for the second integration Player approach to the WDB integration.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb_client	B	28	33	5	0	2	1.18	8.25
wdbinterf_client	F	104	124	20	0	0	1.19	23.85
wdbinterf_driver	F	76	85	11	0	2	1.12	14.54
F-Framework, B-Base, N-New								

Table D.14: Detailed LOC data for the second integration Player approach to the WDB integration.

cost for the Player second integration code are shown in Tables D.14 and D.15. Summarized coupling, LOC, and complexity data by location are shown in Tables D.16, D.17 and D.18.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb_client	B	5	6	1.20	1.20	6.88
wdbinterf_client	F	19	22	1.16	3.47	6.86
wdbinterf_driver	F	11	12	1.09	1.09	13.33

Table D.15: Detailed complexity data for the second integration Player approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	1	2	10	4
Base	0	0	3	2
New	0	0	0	0

Table D.16: Summarized coupling data by location for the second integration Player approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	180	209	31	0	2	1.16	38.28
Base	28	33	5	0	2	1.18	8.25
New	0	0	0	0	0	1.00	0.00

Table D.17: Summarized LOC measurement data by location for the second integration Player approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	30	34	1.13	4.52	8.47
Base	5	6	1.20	1.20	6.88
New	0	0	1.00	0.00	0.00

Table D.18: Summarized complexity data by location for the second integration Player approach to the WDB integration task.

E. P2P WDB INTEGRATION DETAILED DATA

E.1 Baseline

The coupling data obtained from the P2P baseline code are shown in Table E.1. The measures of LOC, McCabe complexity, and complexity change cost for the P2P baseline code are shown in Tables E.2 and E.3. Summarized coupling, LOC, and complexity data by location are shown in Tables E.4, E.5 and E.6.

E.2 First Integration

The coupling data obtained from the P2P first integration code are shown in Table E.7. The measures of LOC, McCabe complexity, and complexity change cost for the P2P first integration code are shown in Tables E.8 and E.9. Summarized coupling, LOC, and complexity data by location are shown in Tables E.10, E.11 and E.12.

E.3 Second Integration

The coupling data obtained from the P2P second integration code are shown in Table E.13. The measures of LOC, McCabe complexity, and complexity change cost

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb-client	F	0	0	12	2
wdb-server	F	0	0	12	4

Table E.1: Detailed coupling data for the baseline P2P approach to the WDB integration.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb-client	F	0	47	47	0	0	1.0	47.0
wdb-server	F	0	76	76	0	0	1.0	76.0

Table E.2: Detailed LOC data for the baseline P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb-client	F	0	10	1.0	10.0	4.7
wdb-server	F	0	19	1.0	19.0	4.0

Table E.3: Detailed complexity data for the baseline P2P approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	0	24	6
New	0	0	0	0

Table E.4: Summarized coupling data by location for the baseline P2P approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.0	0.0
Base	0	123	123	0	0	1.0	123.0
New	0	0	0	0	0	1.0	0.0

Table E.5: Summarized LOC data by location for the baseline P2P approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.0	0.0	0.00
Base	0	29	1.0	29.0	4.24
New	0	0	1.0	0.0	0.00

Table E.6: Summarized complexity data by location for the baseline P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb-client	F	0	0	12	4
wdb-server	F	0	0	12	5

Table E.7: Detailed coupling data for the first integration P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb-client	F	47	74	27	0	0	1.0	27.0
wdb-server	F	76	104	28	0	0	1.0	28.0

Table E.8: Detailed LOC data for the first integration P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb-client	F	10	16	1.60	9.60	4.43
wdb-server	F	19	24	1.26	6.32	6.07

Table E.9: Detailed complexity data for the first integration P2P approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	0	24	9
New	0	0	0	0

Table E.10: Summarized coupling data by location for the first integration P2P approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.00	0.00
Base	123	178	55	0	0	1.45	79.59
New	0	0	0	0	0	1.00	0.00

Table E.11: Summarized LOC data by location for the first integration P2P approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.00	0.00	0.00
Base	29	40	1.38	15.17	5.25
New	0	0	1.00	0.00	0.00

Table E.12: Summarized complexity data by location for the first integration P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
wdb-client	B	0	0	12	6
wdb-server	B	0	0	12	6

Table E.13: Detailed coupling data for the second integration P2P approach to the WDB integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
wdb-client	B	74	90	16	0	8	1.22	29.19
wdb-server	B	104	122	18	0	6	1.17	28.15

Table E.14: Detailed LOC data for the second integration P2P approach to the WDB integration task.

for the P2P second integration code are shown in Tables E.14 and E.15. Summarized coupling, LOC, and complexity data by location are shown in Tables E.16, E.17 and E.18.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
wdb-client	B	16	17	1.06	1.06	27.47
wdb-server	B	24	25	1.04	1.04	27.03

Table E.15: Detailed complexity data for the second integration P2P approach to the WDB integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	0	24	12
New	0	0	0	0

Table E.16: Summarized coupling data by location for the second integration P2P approach to the WDB integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.00	0.00
Base	178	212	34	0	14	1.19	57.17
New	0	0	0	0	0	1.00	0.00

Table E.17: Summarized LOC data by location for the second integration P2P approach to the WDB integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.00	0.00	0.00
Base	40	42	1.05	2.10	27.22
New	0	0	1.00	0.00	0.00

Table E.18: Summarized complexity data by location for the second integration P2P approach to the WDB integration task.

F. SOA PRAGMATICS INTEGRATION DETAILED DATA

F.1 Baseline

The coupling data obtained from the SOA baseline code are shown in Table F.1. The measures of LOC, McCabe complexity, and complexity change cost for the SOA baseline code are shown in Tables F.2 and F.3. Summarized coupling, LOC, and complexity data by location are shown in Tables F.4, F.5 and F.6.

F.2 First Integration

The coupling data obtained from the SOA first integration code are shown in Table F.7. The measures of LOC, McCabe complexity, and complexity change cost for the SOA first integration code are shown in Tables F.8 and F.9. Summarized coupling, LOC, and complexity data by location are shown in Tables F.10, F.11 and F.12.

F.3 Second Integration

The coupling data obtained from the SOA second integration code are shown in Table F.13. The measures of LOC, McCabe complexity, and complexity change cost for the SOA second integration code are shown in Tables F.14 and F.15. Summarized coupling, LOC, and complexity data by location are shown in Tables F.16, F.17 and F.18.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Backend_Messages	F	2	0	3	0
Backend_PragImpl	F	1	0	3	0
Prag_Messages	F	2	0	3	0
Prag_PragmaticsImpl	F	1	0	3	0
GUI_Messages	F	2	0	3	0
GUI_PragImpl	F	1	0	3	0
ServiceUtilities	F	3	0	5	0
IceStormAdapter	F	1	0	11	0
gui	B	0	0	1	0
prag	N	0	0	1	0
backend	N	0	0	1	0

Table F.1: Detailed coupling data for the baseline in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Backend_Messages	F	0	5	5	0	0	1.0	5.0
Backend_PragImpl	F	0	6	6	0	0	1.0	6.0
Prag_Messages	F	0	5	5	0	0	1.0	5.0
Prag_PragmaticsImpl	F	0	7	7	0	0	1.0	7.0
GUI_Messages	F	0	6	6	0	0	1.0	6.0
GUI_PragImpl	F	0	6	6	0	0	1.0	6.0
ServiceUtilities	F	0	46	46	0	0	1.0	46.0
IceStormAdapter	F	0	55	55	0	0	1.0	55.0
gui	B	0	4	4	0	0	1.0	4.0
prag	N	0	4	4	0	0	1.0	4.0
backend	N	0	4	4	0	0	1.0	4.0

Table F.2: Detailed LOC data for the baseline in the SOA approach to the Pragmatics integration.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Backend_Messages	F	0	5	1.0	5.0	1.00
Backend_PragImpl	F	0	6	1.0	6.0	1.00
Prag_Messages	F	0	5	1.0	5.0	1.00
Prag_PragmaticsImpl	F	0	7	1.0	7.0	1.00
GUI_Messages	F	0	5	1.0	5.0	1.20
GUI_PragImpl	F	0	6	1.0	6.0	1.00
ServiceUtilities	F	0	13	1.0	13.0	3.54
IceStormAdapter	F	0	13	1.0	13.0	4.23
gui	B	0	1	1.0	1.0	4.00
prag	N	0	1	1.0	1.0	4.00
backend	N	0	1	1.0	1.0	4.00

Table F.3: Detailed complexity data for the baseline in the SOA approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	13	0	34	0
Base	0	0	1	0
New	0	0	2	0

Table F.4: Summarized coupling data by location for the baseline in the SOA approach to the Pragmatics integration.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	136	136	0	0	1.0	136.0
Base	0	4	4	0	0	1.0	4.0
New	0	8	8	0	0	1.0	8.0

Table F.5: Summarized LOC data by location for the baseline in the SOA approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	60	1.0	60.0	2.27
Base	0	1	1.0	1.0	4.00
New	0	2	1.0	2.0	4.00

Table F.6: Summarized complexity data by location for the baseline in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Backend_Messages	F	2	0	3	2
Backend_PragImpl	F	1	0	3	2
Prag_Messages	F	2	0	3	4
Prag_PragmaticsImpl	F	1	0	3	2
GUI_Messages	F	2	0	3	2
GUI_PragImpl	F	1	0	3	1
ServiceUtilities	F	3	0	5	0
IceStormAdapter	F	1	0	11	0
gui	B	0	0	1	0
prag	N	0	0	1	0
backend	N	0	0	1	0

Table F.7: Detailed coupling data for the first integration in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Backend_Messages	F	5	8	3	0	0	1.60	4.80
Backend_PragImpl	F	6	10	4	0	0	1.67	6.67
Prag_Messages	F	5	14	9	0	0	2.80	25.20
Prag_PragmaticsImpl	F	7	15	8	0	0	2.14	17.14
GUI_Messages	F	6	10	4	0	0	1.67	6.67
GUI_PragImpl	F	6	10	4	0	0	1.67	6.67
ServiceUtilities	F	46	46	0	0	0	1.00	0.00
IceStormAdapter	F	55	55	0	0	0	1.00	0.00
gui	B	4	5	1	0	0	1.25	1.25
prag	N	4	4	0	0	0	1.00	0.00
backend	N	4	4	0	0	0	1.00	0.00

Table F.8: Detailed LOC data for the first integration in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Backend_Messages	F	5	7	1.40	2.80	1.71
Backend_PragImpl	F	6	7	1.17	1.17	5.71
Prag_Messages	F	5	9	1.80	7.20	3.50
Prag_PragmaticsImpl	F	7	9	1.29	2.57	6.67
GUI_Messages	F	5	7	1.40	2.80	2.38
GUI_PragImpl	F	6	7	1.17	1.17	5.71
ServiceUtilities	F	13	13	1.00	0.00	0.00
IceStormAdapter	F	13	13	1.00	0.00	0.00
gui	B	1	1	1.00	0.00	0.00
prag	N	1	1	1.00	0.00	0.00
backend	N	1	1	1.00	0.00	0.00

Table F.9: Detailed complexity data for the first integration in the SOA approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	13	0	34	13
Base	0	0	1	0
New	0	0	2	0

Table F.10: Summarized coupling data by location for the first integration in the SOA approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	136	168	32	0	0	1.24	39.68
Base	4	5	1	0	0	1.25	1.25
New	8	8	0	0	0	1.00	0.00

Table F.11: Summarized LOC data by location for the first integration in the SOA approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	60	72	1.2	14.4	
Base	1	1	1.0	1.0	
New	2	2	1.0	1.0	

Table F.12: Summarized complexity data by location for the first integration in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Backend_Messages	F	2	0	3	4
Backend_PragImpl	F	1	0	3	2
Prag_Messages	F	2	0	3	4
Prag_PragmaticsImpl	F	1	0	3	2
GUI_Messages	F	2	0	3	7
GUI_PragImpl	F	1	0	3	3
ServiceUtilities	F	3	0	5	0
IceStormAdapter	F	1	0	11	0
gui	B	0	0	1	0
prag	N	0	0	1	0
backend	N	0	0	1	0

Table F.13: Detailed coupling data for the second integration in the SOA approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Backend_Messages	F	8	10	2	0	0	1.25	2.50
Backend_PragImpl	F	10	10	0	0	0	1.00	0.00
Prag_Messages	F	14	18	4	0	0	1.29	5.14
Prag_PragmaticsImpl	F	15	20	5	0	1	1.33	8.00
GUI_Messages	F	10	12	3	1	0	1.20	4.80
GUI_PragImpl	F	10	12	2	0	0	1.20	2.40
ServiceUtilities	F	46	46	0	0	0	1.00	0.00
IceStormAdapter	F	55	55	0	0	0	1.00	0.00
gui	B	5	6	1	0	0	1.20	1.20
prag	N	4	4	0	0	0	1.00	0.00
backend	N	4	4	0	0	0	1.00	0.00

Table F.14: Detailed LOC data for the second integration in the SOA approach to the Pragmatics integration.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Backend_Messages	F	7	8	1.14	1.14	2.19
Backend_PragImpl	F	7	7	1.00	0.00	0.00
Prag_Messages	F	9	12	1.33	3.00	1.71
Prag_PragmaticsImpl	F	9	11	1.22	4.91	1.63
GUI_Messages	F	7	9	1.29	3.11	1.54
GUI_PragImpl	F	7	8	1.40	1.75	1.37
ServiceUtilities	F	13	13	1.17	0.00	0.00
IceStormAdapter	F	13	13	1.00	0.00	0.00
gui	B	1	1	1.00	0.00	0.00
prag	N	1	1	1.00	0.00	0.00
backend	N	1	1	1.00	0.00	0.00

Table F.15: Detailed complexity data for the second integration in the SOA approach to the Pragmatics integration.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	13	0	34	22
Base	0	0	1	0
New	0	0	2	0

Table F.16: Summarized coupling data by location for the second integration in the SOA approach to the Pragmatics integration.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	168	183	16	1	1	1.09	17.44
Base	5	6	1	0	0	1.20	1.20
New	8	8	0	0	0	1.00	0.00

Table F.17: Summarized LOC data by location for the second integration in the SOA approach to the Pragmatics integration.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	72	81	1.13	10.17	1.71
Base	1	1	1.00	1.00	0.00
New	2	2	1.00	1.00	0.00

Table F.18: Summarized complexity data by location for the second integration in the SOA approach to the Pragmatics integration.

G. PLAYER PRAGMATICS INTEGRATION DETAILED DATA

G.1 Baseline

The coupling data obtained from the Player baseline code are shown in Table G.1. The measures of LOC, McCabe complexity. and complexity change cost for the Player baseline code are shown in Tables G.2 and G.3. Summarized coupling, LOC, and complexity data by location are shown in Tables G.4, G.5 and G.6.

G.2 First Integration

The coupling data obtained from the Player first integration code are shown in Table G.7. The measures of LOC, McCabe complexity, and complexity change cost for the Player first integration code are shown in Tables G.8 and G.9. Summarized coupling, LOC, and complexity data by location are shown in Tables G.10, G.11 and G.12.

G.3 Second Integration

The coupling data obtained from the Player second integration code are shown in Table G.13. The measures of LOC, McCabe complexity, and complexity change cost for the Player second integration code are shown in Tables G.14 and G.15. Summarized coupling, LOC, and complexity data by location are shown in Tables G.16, G.17 and G.18.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muribackendinterf_client	F	1	1	2	1
muribackendinterf_driver	F	0	1	4	0
muripraginterf_client	F	1	1	2	1
muripraginterf_driver	F	0	1	4	0
muriguiinterf_client	F	1	1	2	1
muriguiinterf_driver	F	0	1	4	0
muri_backend	B	0	0	4	3
muri_pragmatics	B	0	0	4	3
muri_gui	B	0	0	6	5

Table G.1: Detailed coupling data for the baseline Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muribackendinterf_client	F	0	9	9	0	0	1.0	9.0
muribackendinterf_driver	F	0	15	15	0	0	1.0	15.0
muripraginterf_client	F	0	13	13	0	0	1.0	13.0
muripraginterf_driver	F	0	15	15	0	0	1.0	15.0
muriguiinterf_client	F	0	13	13	0	0	1.0	13.0
muriguiinterf_driver	F	0	15	15	0	0	1.0	15.0
muri_backend	B	0	18	18	0	0	1.0	18.0
muri_pragmatics	B	0	18	18	0	0	1.0	18.0
muri_gui	B	0	32	32	0	0	1.0	32.0

Table G.2: Detailed LOC data for the baseline Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muribackendinterf_client	F	0	5	1.0	5.0	1.80
muribackendinterf_driver	F	0	7	1.0	7.0	2.14
muripraginterf_client	F	0	6	1.0	6.0	2.17
muripraginterf_driver	F	0	7	1.0	7.0	2.14
muriguiinterf_client	F	0	6	1.0	6.0	2.17
muriguiinterf_driver	F	0	7	1.0	7.0	2.14
muri_backend	B	0	4	1.0	4.0	4.50
muri_pragmatics	B	0	4	1.0	4.0	4.50
muri_gui	B	0	6	1.0	6.0	5.33

Table G.3: Detailed complexity data for the baseline Player approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	3	6	28	11
Base	0	0	4	3
New	0	0	0	0

Table G.4: Summarized coupling data by location for the baseline Player approach to the Pragmatics integration.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	130	130	0	0	1.0	130.0
Base	0	18	18	0	0	1.0	18.0
New	0	0	0	0	0	1.0	0.0

Table G.5: Summarized LOC data by location for the baseline Player approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	48	1.0	48.0	2.7
Base	0	4	1.0	4.0	4.5
New	0	0	1.0	0.0	0.0

Table G.6: Summarized complexity data by location for the baseline Player approach to the Pragmatics integration.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muribackendinterf_client	F	1	1	2	1
muribackendinterf_driver	F	0	1	5	0
muripraginterf_client	F	1	1	2	1
muripraginterf_driver	F	0	1	5	0
muriguiinterf_client	F	1	1	2	1
muriguiinterf_driver	F	0	1	5	0
muri_backend	B	0	0	4	3
muri_pragmatics	B	0	0	4	3
muri_gui	B	0	0	6	5

Table G.7: Detailed coupling data for the first integration Player approach to the Pragmatics integration.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muribackendinterf_client	F	9	12	3	0	0	1.33	4.00
muribackendinterf_driver	F	15	18	3	0	0	1.20	3.60
muripraginterf_client	F	13	37	21	0	0	2.85	59.77
muripraginterf_driver	F	15	34	21	0	0	2.27	47.60
muriguiinterf_client	F	13	31	18	0	0	2.38	42.92
muriguiinterf_driver	F	15	24	9	0	0	1.60	14.40
muri_backend	B	18	26	8	0	0	1.44	11.56
muri_pragmatics	B	18	28	10	0	0	1.56	15.56
muri_gui	B	32	50	18	0	0	1.56	28.13

Table G.8: Detailed LOC data for the first integration Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muribackendinterf_client	F	5	6	1.20	1.20	3.33
muribackendinterf_driver	F	7	8	1.14	1.14	3.15
muripraginterf_client	F	6	10	1.67	6.67	8.97
muripraginterf_driver	F	7	9	1.29	2.57	18.51
muriguiinterf_client	F	6	11	1.83	9.17	4.68
muriguiinterf_driver	F	7	8	1.14	1.14	12.60
muri_backend	B	4	4	1.00	0.00	0.00
muri_pragmatics	B	4	5	1.25	1.25	12.44
muri_gui	B	6	11	1.83	9.17	3.07

Table G.9: Detailed complexity data for the first integration Player approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	3	6	31	11
Base	0	0	4	3
New	0	0	0	0

Table G.10: Summarized coupling data by location for the first integration Player approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	130	234	103	0	0	1.80	185.40
Base	18	26	8	0	0	1.44	11.56
New	0	0	0	0	0	1.00	0.00

Table G.11: Summarized LOC data by location for the first integration Player approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	10	16	1.60	9.6	19.31
Base	4	4	1.00	4.0	0.00
New	0	0	1.00	0.0	0.00

Table G.12: Summarized complexity data by location for the first integration Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muribackendinterf_client	F	1	1	2	1
muribackendinterf_driver	F	0	1	5	0
muripraginterf_client	F	1	1	2	1
muripraginterf_driver	F	0	1	5	0
muriguiinterf_client	F	1	1	2	1
muriguiinterf_driver	F	0	1	5	0
muri_backend	B	0	0	4	3
muri_pragmatics	B	0	0	4	3
muri_gui	B	0	0	6	5

Table G.13: Detailed coupling data for the second integration Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muribackendinterf_client	F	12	15	3	0	0	1.25	3.75
muribackendinterf_driver	F	18	21	3	0	0	1.17	3.50
muripraginterf_client	F	37	40	3	0	0	1.08	3.24
muripraginterf_driver	F	34	36	2	0	0	1.06	2.12
muriguiinterf_client	F	31	34	3	0	0	1.10	3.29
muriguiinterf_driver	F	24	30	6	0	0	1.25	7.50
muri_backend	B	26	26	0	0	0	1.00	0.00
muri_pragmatics	B	28	33	5	0	0	1.18	5.89
muri_gui	B	50	50	0	0	0	1.00	0.00

Table G.14: Detailed LOC data for the second integration Player approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muribackendinterf_client	F	6	7	1.17	1.17	3.21
muribackendinterf_driver	F	8	9	1.13	1.13	3.11
muripraginterf_client	F	10	11	1.10	1.10	2.95
muripraginterf_driver	F	9	10	1.11	1.11	1.91
muriguiinterf_client	F	11	12	1.09	1.09	3.02
muriguiinterf_driver	F	8	9	1.13	1.13	6.67
muri_backend	B	4	4	1.00	0.00	0.00
muri_pragmatics	B	5	6	1.20	1.20	4.91
muri_gui	B	11	11	1.00	0.00	0.00

Table G.15: Detailed complexity data for the second integration Player approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	3	0	31	11
Base	0	0	4	3
New	0	0	0	0

Table G.16: Summarized coupling data by location for the second integration Player approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	234	259	25	0	0	1.11	27.75
Base	26	26	0	0	0	1.00	0.00
New	0	0	0	0	0	1.00	0.00

Table G.17: Summarized LOC data by location for the second integration Player approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	68	75	1.10	7.70	3.60
Base	4	4	1.00	0.00	0.00
New	0	0	1.00	0.00	0.00

Table G.18: Summarized complexity data by location for the second integration Player approach to the Pragmatics integration task.

H. P2P PRAGMATICS INTEGRATION DETAILED DATA

H.1 Baseline

The coupling data obtained from the P2P baseline code are shown in Table H.1. The measures of LOC, McCabe complexity, and complexity change cost for the P2P baseline code are shown in Tables H.2 and H.3. Summarized coupling, LOC, and complexity data by location are shown in Tables H.4, H.5 and H.6.

H.2 First Integration

The coupling data obtained from the P2P first integration code are shown in Table H.7. The measures of LOC, McCabe complexity, and complexity change cost for the P2P first integration code are shown in Tables H.8 and H.9. Summarized coupling, LOC, and complexity data by location are shown in Tables H.10, H.11 and H.12.

H.3 Second Integration

The coupling data obtained from the P2P second integration code are shown in Table H.13. The measures of LOC, McCabe complexity, and complexity change cost

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muri-backend	B	0	1	10	3
muri-gui	B	0	1	8	2
muri-prag	B	0	2	10	5

Table H.1: Detailed coupling data for the baseline P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muri-backend	B	0	67	67	0	0	1.0	67.0
muri-gui	B	0	35	35	0	0	1.0	35.0
muri-prag	B	0	94	94	0	0	1.0	94.0

Table H.2: Detailed LOC data for the baseline P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muri-backend	B	0	17	1.0	17.0	3.94
muri-gui	B	0	9	1.0	9.0	3.89
muri-prag	B	0	22	1.0	22.0	4.27

Table H.3: Detailed complexity data for the baseline P2P approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	4	28	10
New	0	0	0	0

Table H.4: Summarized coupling data by location for the baseline P2P approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.0	0.0
Base	0	196	196	0	0	1.0	196.0
New	0	0	0	0	0	1.0	0.0

Table H.5: Summarized LOC data by location for the baseline P2P approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.0	0.0	0.00
Base	0	48	1.0	48.0	4.08
New	0	0	1.0	0.0	0.00

Table H.6: Summarized complexity data by location for the baseline P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muri-backend	B	0	1	10	5
muri-prag	B	0	1	8	3
muri-gui	B	0	2	10	9

Table H.7: Detailed coupling data for the first integration P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muri-backend	B	67	72	5	0	0	1.07	5.37
muri-prag	B	35	45	10	0	0	1.29	12.86
muri-gui	B	94	109	15	0	0	1.16	17.39

Table H.8: Detailed LOC data for the first integration P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muri-backend	B	17	18	1.06	1.06	5.07
muri-prag	B	9	11	1.22	2.44	5.26
muri-gui	B	22	25	1.14	3.41	5.10

Table H.9: Detailed complexity data for the first integration P2P approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	4	28	17
New	0	0	0	0

Table H.10: Summarized coupling data by location for the first integration P2P approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.00	0.00
Base	196	226	30	0	0	1.15	34.59
New	0	0	0	0	0	1.00	0.00

Table H.11: Summarized LOC measurement data by location for the first integration P2P approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.00	0.00	0.00
Base	48	54	1.13	6.75	5.12
New	0	0	1.00	0.00	0.00

Table H.12: Summarized complexity data by location for the first integration P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
muri-backend	B	0	1	10	6
muri-prag	B	0	1	8	6
muri-gui	B	0	2	10	11

Table H.13: Detailed coupling data for the second integration source code in P2P approach to the Pragmatics integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
muri-backend	B	72	86	27	13	15	1.19	65.69
muri-prag	B	45	64	25	6	12	1.42	61.16
muri-gui	B	109	120	56	45	20	1.10	133.21

Table H.14: Detailed LOC data for the second integration P2P approach to the Pragmatics integration task.

for the P2P second integration code are shown in Tables H.14 and H.15. Summarized coupling, LOC, and complexity data by location are shown in Tables H.16, H.17 and H.18.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
muri-backend	B	18	22	1.22	4.89	13.44
muri-prag	B	11	14	1.27	3.82	16.02
muri-gui	B	25	31	1.24	7.44	17.90

Table H.15: Detailed complexity data for the second integration P2P approach to the Pragmatics integration task.

Location	$s_a(\mathcal{L})$	$d_a(\mathcal{L})$	$s_e(\mathcal{L})$	$d_e(\mathcal{L})$
Framework	0	0	0	0
Base	0	4	28	23
New	0	0	0	0

Table H.16: Summarized coupling data by location for the second integration P2P approach to the Pragmatics integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Framework	0	0	0	0	0	1.00	0.00
Base	226	270	108	64	47	1.19	261.64
New	0	0	0	0	0	1.00	0.00

Table H.17: Summarized LOC data by location for the second integration P2P approach to the Pragmatics integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Framework	0	0	1.00	0.00	0.00
Base	54	67	1.24	16.13	16.22
New	0	0	1.00	0.00	0.00

Table H.18: Summarized complexity data by location for the second integration P2P approach to the Pragmatics integration task.

I. PLAYER MODULE INTEGRATION DETAILED DATA

I.1 Baseline

The measures of LOC, McCabe complexity, and complexity change cost for the Player module integration baseline code are shown in Tables I.1 and I.2. Summarized coupling, LOC, and complexity data by location are shown in Table I.3 and I.4.

I.2 First Integration

The measures of LOC, McCabe complexity, and complexity change cost for the Player module first stage integration code are shown in Tables I.5 and I.6. Summarized coupling, LOC, and complexity data by location are shown in Table I.7 and I.8.

I.3 Second Integration

The measures of LOC, McCabe complexity, and complexity change cost for the second stage Player module integration code are shown in Tables I.9 and I.10. Summarized coupling, LOC, and complexity data by location are shown in Table I.11 and I.12.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
consumer	B	0	8	8	0	0	1.00	8.00
p2d-consumer	F	0	10	10	0	0	1.00	10.00
p2d-mediator-consumer	F	0	18	18	0	0	1.00	18.00
provider	F	0	11	11	0	0	1.00	11.00
p2d-provider	F	0	56	56	0	0	1.00	56.00
p2d-mediator-provider	F	0	49	49	0	0	1.00	49.00
p2d-mediator	F	0	22	22	0	0	1.00	22.00
F-Framework, B-Base, N-New								

Table I.1: Detailed LOC data for the baseline SOA approach to the Player module integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
consumer	B	0	1	1.00	1.00	8.00
p2d-consumer	F	0	2	1.00	2.00	5.00
p2d-mediator-consumer	F	0	2	1.00	2.00	9.00
provider	F	0	1	1.00	1.00	11.00
p2d-provider	F	0	15	1.00	15.00	3.73
p2d-mediator-provider	F	0	6	1.00	6.00	8.17
p2d-mediator	F	0	2	1.00	2.00	11.00

Table I.2: Detailed complexity data for the baseline SOA approach to the Player module integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Base	0	8	8	0	0	1.00	8.00
Framework	0	166	166	0	0	1.00	166.00
New	0	0	0	0	0	1.00	0.00

Table I.3: Summarized LOC data by location for the baseline SOA approach to the Player module integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Base	0	1	1.00	1.00	8.00
Framework	0	28	1.00	28.00	6.00
New	0	0	1.00	0.00	1.00

Table I.4: Summarized complexity data by location for the baseline SOA approach to the Player module integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
consumer	B	8	10	2	0	0	1.25	2.50
p2d-consumer	F	10	10	0	0	0	1.00	0.00
p2d-mediator-consumer	F	18	28	10	0	0	1.56	15.56
provider	F	11	11	0	0	0	1.00	0.00
p2d-provider	F	56	58	2	0	1	1.04	3.11
p2d-mediator-provider	F	49	74	25	0	0	1.51	37.76
p2d-mediator	F	22	22	0	0	0	1.00	0.00
F-Framework, B-Base, N-New								

Table I.5: Detailed LOC data for the first stage SOA approach to the Player module integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
consumer	B	1	1	1.00	0.00	0.00
p2d-consumer	F	2	2	1.00	0.00	0.00
p2d-mediator-consumer	F	2	3	1.50	1.50	10.37
provider	F	1	1	1.00	0.00	0.00
p2d-provider	F	15	15	1.00	0.00	0.00
p2d-mediator-provider	F	6	7	1.17	1.17	32.36
p2d-mediator	F	2	2	1.00	0.00	0.00

Table I.6: Detailed complexity data for the first stage SOA approach to the Player module integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Base	8	10	2	0	0	1.25	2.50
Framework	174	203	37	0	1	1.22	46.36
New	0	0	0	0	0	1.00	0.00

Table I.7: Summarized LOC data by location for the first stage SOA approach to the Player module integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Base	1	1	1.00	0.00	0.00
Framework	28	30	1.07	2.14	21.66
New	0	0	1.00	0.00	0.00

Table I.8: Summarized complexity data by location for the first stage SOA approach to the Player module integration task.

Software Element	$\mathcal{L}$	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
consumer	B	10	12	2	0	0	1.20	2.40
p2d-consumer	F	10	10	0	0	0	1.00	0.00
p2d-mediator-consumer	F	28	38	10	0	0	1.36	13.57
provider	F	11	11	0	0	0	1.00	0.00
p2d-provider	F	58	60	2	0	1	1.03	3.10
p2d-mediator-provider	F	74	96	22	0	0	1.30	28.54
p2d-mediator	F	22	22	0	0	0	1.00	0.00
F-Framework, B-Base, N-New								

Table I.9: Detailed LOC data for the second stage SOA approach to the Player module integration task.

Software Element	$\mathcal{L}$	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
consumer	B	1	1	1.00	0.00	0.00
p2d-consumer	F	2	2	1.00	0.00	0.00
p2d-mediator-consumer	F	3	4	1.33	1.33	10.18
provider	F	1	1	1.00	0.00	0.00
p2d-provider	F	15	15	1.00	0.00	0.00
p2d-mediator-provider	F	7	8	1.14	1.14	24.97
p2d-mediator	F	2	2	1.00	0.00	0.00

Table I.10: Detailed complexity data for the second stage SOA approach to the Player module integration task.

Location	$\lambda_{pre}(\mathcal{L})$	$\lambda_{post}(\mathcal{L})$	$\Gamma_A(\mathcal{L})$	$\Gamma_R(\mathcal{L})$	$\Gamma_M(\mathcal{L})$	$\hat{\lambda}(\mathcal{L})$	$\bar{\Gamma}(\mathcal{L})$
Base	10	12	2	0	0	1.20	2.40
Framework	203	237	34	0	1	1.17	40.95
New	0	0	0	0	0	1.00	0.00

Table I.11: Summarized LOC data by location for the second stage SOA approach to the Player module integration task.

Location	$v_{g_{pre}}(\mathcal{L})$	$v_{g_{post}}(\mathcal{L})$	$\hat{v}_g(\mathcal{L})$	$\bar{v}_g(\mathcal{L})$	$E(\mathcal{L})$
Base	1	1	1.00	0.00	0.00
Framework	30	32	1.07	2.14	19.14
New	0	0	1.00	0.00	0.00

Table I.12: Summarized complexity data by location for the second stage SOA approach to the Player module integration task.

REFERENCES

J. S. Albus. 4D/RCS: A Reference Model Architecture for Intelligent Unmanned Ground Vehicles. In *Proceedings of the SPIE 16th Annual International Symposium on AerospaceDefense Sensing, Simulation, and Controls*, April 2002.

R. C. Arkin. Motor Schema-Based Mobile Robot Navigation. *International Journal of Robotic Research*, 8(4):92–112, 1989.

M. Barbacci, T. H. Longstaff, M. H. Klein, and C. B. Weinstock. Quality attributes. Technical Report CMU/SEI-95-TR-021, Carnegie Mellon University, Software Engineering Institute, Dec 1995.

D. M. Beazley. SWIG Website Documentation, 2007.

D. Blank, D. Kumar, L. Meeden, and H. Yanco. Pyro: A Python-based Versatile Programming Environment for Teaching Robotics. *ACM Journal on Educational Resourcs in Computing (JERIC)*, 4(3), Sep 2004.

D. Blank, D. Kumar, L. Meeden, and H. Yanco. The Pyro Toolkit for AI and Robotics. *AI Magazine*, 27(1):39–50, 2006.

D. S. Blank, L. Meeden, and D. Kumar. Python robotics: An Environment for Exploring Robotics Beyond LEGOs. In SIGCSE, editor, *ACM Special Interest Group: Computer Science Education Conference*, 2003.

R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. G. Slack. Experiences with an Architecture for Intelligent Reactive Agents. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1995.

G. Booch. Handbook of Software Architecture. Online, 2010. URL `{http://www.handbookofsoftwarearchitecture.com/index.jsp?page=Patterns}%`. Requires registration and login for access.

L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3(1):65–117, Mar 1998.

L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25 (1):91–121, Jan-Feb 1999.

R. A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, Mar 1986.

W. Buchholz. *Planning a computer system: Project Stretch.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1962. ISBN B0000CLCYO.

CANBus, 1991. CAN Specification Version 2.0, September 1991. URL `http://www.semiconductors.bosch.de/pdf/can2spec.pdf`.

L. Chaimowicz, A. Cowley, V. Sabella, and C. J. Taylor. ROCI: A Distributed Framework for Multi-Robot Perception and Control. In *Proceedings of the 2003 IEEE/RJS International Conference on Intelligent Robots and Systems, Las Vegas, NV*, pages 266–271, 2003.

S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.295895.

P. Clements, R. Katzman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies.* Addison-Wesley, 2002. ISBN 0-201-70482-X.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond.* Addison-Wesley, Pearson Education, 2004. ISBN 0-201-70372-6.

E. Colon, H. Sahli, and Y. Baudoin. CoRoBa, a multi mobile robot control and simulation framework. *International Journal of Advanced Robotic Systems*, 3(1): 073–078, 2006. ISSN 1729-8806.

C. Côté, D. Létourneau, F. Michaud, J. -M. Valin, Y. Brosseau, C. Raïevasky, M. Lemay, and V. Tran. Code Reusability Tools for Programming Mobile Robots. In *Proceedings IEEE/RJS International Conference on Intelligent Robots and Systems*, pages 1820–1825, 2004.

G. Dedene and M. Snoeck. M.E.R.O.D.E.: A Model-driven Entity-Relationship Object-oriented Development method. *ACM SIGSOFT Software Engineering Notes*, 13:51–61, 1994.

Department of Defense. JAUS Strategic Plan, version 1.5, Apr 2005.

Department of Defense. Joint Architecture for Unmanned Systems, Reference Architecture Specification, version 3.2, Aug 2004.

DoD Business Transformation Agency. Business Enterprise Architecture (BEA) 7.0. Online, 2010. URL `http://www.bta.mil/products/BEA_7_0/html_files/home.html`.

DoD Deputy CIO. The DoDAF Architecture Framework Version 2.0. Online, 2009. URL `http://www.cio-nii.defense.gov/sites/dodaf20/`.

Douglas Schmidt. Real-time CORBA with TAO (The ACE Orb), Sep 2010. URL `http://www.cs.wustl.edu/~schmidt/TAO.html`.

T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Pearson Education, 2009a. ISBN 978-0-13-185858-9.

T. Erl. SOA Patterns. Online, 2010. URL `http://www.soapatterns.org/`.

T. Erl. *SOA Design Patterns*. Prentice Hall/PearsonPTR, 2009b. ISBN 0136135161.

W. J. Fabrycky and B. S. Blanchard. *Systems Engineering Analysis*. Prentice Hall, 3rd edition, 1998. ISBN 978-0131350472.

N. Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar 1994.

N. Fenton and S. Pfleeger. *Software Metrics — A Rigorous and Practical Approach*. Brooks Cole Publishing Company, 1998. ISBN 0534954251.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Pearson Education, 2003. ISBN 0321127420.

M. Galic, J. Adams, J. A. Bell, R. Disney, V.-M. Kanerva, S. Matulevich, K. Rebman, and P. Spaas. Patterns: Applying Pattern Approaches. Online, 2003. URL `http://www.redbooks.ibm.com/redbooks/pdfs/sg246805.pdf`.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse in Object-Oriented Designs. In O. Nierstrasz, editor, *Proceedings of ECOOP'93*, Berlin, Germany, 1993. Springer-Verlag.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-201-63361-2.

B. Gerkey, R. Vaughan, K. Sty, A. Howard, G. Sukhatme, and M. Mataric. Most valuable player: A robot device server for distributed control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2001.

B. Gerkey, R. Vaughan, and A. Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, pages 317–323, 2003.

M. Gertz. A Human-Machine Interface For Reconfigurable Sensor-Based Control Systems. In *Proceedings of the 1993 AIAA Conference on Space Programs and Technologies*. AIAA, Sep 1993.

N. S. Gill and Balkishan. Dependency and Interaction Oriented Complexity Metrics of Component-Based Systems. *ACM SIGSOFT Software Engineering Notes*, 33(2): 1–5, Jan 2008.

T. Gorton and B. Mikhak. A tangible architecture for creating modular, subsumption-based robot control systems. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1469–1472, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-703-6.

B. Hall. Beej's Guide to Network Programming: Introduction, Mar 2009a. URL `http://beej.us/guide/bgnet/output/html/multipage/intro.html#copyright`.

B. Hall. Beej's Guide to Network Programming: Using Internet Sockets, Mar 2009b. URL `http://beej.us/guide/bgnet/output/html/multipage/clientserver.html`.

S. M. Henry and D. G. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

H. Hofmeister and G. Wirtz. Supporting service oriented design with metrics. *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 191–200, Dec 2008.

G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison Wesley, Pearson Education, 2004. ISBN 0321200683.

G. Hohpe and B. Woolf. Enterprise Integration Patterns, Pattern Catalog. Online, 2003. URL `http://www.eaipatterns.com/toc.html`.

B. Hulin. A Software-Architecture for Sensor Integration in Advanced Robotic Systems. In *Photomec, Second European Workshop on Photonics in Mechanical and Industrial Processes*, Feb 2003.

IEEE. ANSI/IEEE Standard 1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems. Technical report, ANSI/IEEE, 2000.

iRobot. Mobility Integration Software User's Guide, 2002.

F. Jiao, C. Hu, and C. Zhao. A Software Complexity Metric for SCA Specification. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 481–484. IEEE, Dec 2008. doi: http://doi.acm.org/10.1109/CSSE.2008.615.

N. Josuttis. *SOA In Practice*. O'Reilly, 2007. ISBN 9780596529550.

JTADG. Joint Technical Architecture, Version 3.1. Technical report, Joint Technical Architecture Development Group, Oct 2003.

C. Kaner and W. P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? In *10th International Software Metrics Symposium, Metrics 2004*, 2004.

C. Kapoor. *A Reusable Operational Software Architecture for Advanced Robotics*. PhD thesis, The University of Texas at Austin, Dec 1996.

V. Kawadia and P. R. Kumar. A cautionary perspective on cross layer. *IEEE WIRELESS COMMUNICATION MAGAZINE*, 12:3–11, 2005.

H. Kenn, S. Carpin, M. Pfingsthorn, B. Hepes, C. Ciocov, and A. Birk. FAST-Robots: a rapid-prototyping framework for intelligent mobile robotics. In *Artificial Intelligence and Applications Conference*, 2003.

L. Kharb and R. Singh. Complexity metrics for component-oriented software systems. *SIGSOFT Software Engineering Notes*, 33(2):1–3, Mar 2008.

P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.469759.

C. Lilienthal. Architectural Complexity of Large-Scale Software Systems. *European Conference on Software Maintenance and Reengineering*, pages 17–26, Feb 2009.

T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

Microsoft Corporation. COM: Component Object Technologies, Sep 2010. URL `http://www.microsoft.com/com/default.mspx`.

E. E. Mills. Software metrics. Technical Report SEI Curriculum Module SEI-CM-12-1.1, Carnegie Mellon University, Software Engineering Institute, Dec 1988.

R. R. Murphy. *An Introduction to AI Robotics*. The MIT Press, 1st edition, Nov 2000. ISBN 978-0262133838.

N. J. Nilsson. Shakey The Robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr 1984.

OASIS. Organization for the Advancement of Structured Information Systems, Sep 2010. URL `http://www.oasis-open.org/committees/tc_cat.php?cat=soa`.

OASIS. OASIS Reference Architecture Foundation for Service Oriented Architecture 1.0, Committee Draft 2.0. Online, 2009. URL `http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.pdf`.

Object Management Group. CORBA Basics, Dec 2007. URL `http://www.omg.org/gettingstarted/corbafaq.htm`.

Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. Technical report, Object Management Group, 2008. URL `http://www.omg.org/spec/CORBA/3.1`.

OMG. Data Distribution Service for Real-time Systems, Version 1.2. Technical Report OMG Available Specification formal/07-01-01, Object Management Group, Jan 2007.

Oracle. Remote Method Invocation Home, Sep 2010. URL `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.htm%l`.

M. Perepletchikov, C. Ryan, K. Frampton, and H. W. Schmidt. A Formal Model of Service-Oriented Design Structure. In *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)*, volume 0, pages 71–80, 2007a.

M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari. Coupling Metrics for Predicting Maintainability in Service-Oriented Designs. In *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)*, 2007b.

Profibus, 2002. PROFIBUS Technology and Application—System Description, 2002. URL `http://www.profibus.com/nc/downloads/downloads/profibus-technology-and-%application-system-description/display/`.

M. Quiqley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *International Conference on Robotics and Automation, Workshop on Open-Source Robotics, 2009*, May 2009.

F. Riguzzi. A Survey of Software Metrics. *DEIS Technical Report*, DEIS-LIA-96-010: 1–32, Nov 1996.

D. Rud, A. Schmietendorf, and R. R. Dumke. Product metrics for service-oriented infrastructures. *IWSM/MetriKon 2006*, pages 1–14, Aug 2006.

SEI. Software Engineering Institute Architecture Portal, 2008. URL `http://www.sei.cmu.edu`.

M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *In Proceedings of the Third International Software Product Line Conference (SPLC)*, 2004.

J. F. Sowa and J. A. Zachman. Extending and Formalizing the Framework for Information Systems Architecture. *IBM Systems Journal*, 31(3), 1992.

The Hillside Group. Patterns. Online, 2010. URL `http://www.hillside.net/patterns`. Large pattern repository for patterns from all sorts of disciplines, including software architecture.

The Open Group. The Open Group Architectural Framework (TOGAF), version 9. Technical report, The Open Group, 2010a. URL `http://www.opengroup.org/architecture/togaf9-doc/arch/`.

The Open Group. TOGAF Version 9 Enterprise Edition, Core Concepts, Sep 2010b. URL `http://www.opengroup.org/architecture/togaf9-doc/arch/`.

U.S. Department of Defense. FY2009-2034 Unmanned Systems Integrated Roadmap. pages 1–210, Apr 2009.

US Dept of the Treasury CIO Council. Treasury Enterprise Architecture Framework. Online, 2001. URL `http://www.eaframeworks.com/TEAF/teaf.doc`.

U.S. Office of Management and Budget. Federal Enterprise Architecture (FEA), 2010. URL `http://www.whitehouse.gov/omb/e-gov/fea/`.

R. Vaughan, B. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2121–2427, Oct 2003. Also as Technical Report CRES-03-009.

T. Vernazza, G. Granatella, G. Succi, L. Benedicenti, and M. Mintchev. Defining metrics for software components. *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics*, XI:16–23, May 2000.

R. Volpe, I. A. D. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. CLARAty: Coupled Layer Architecture for Robotic Autonomy. Technical report, Jet Propulsion Laboratory, Dec 2000.

Y. Wang. On the Cognitive Complexity of Software and its Quantification and Formal Measurement. *International Journal of Software Science and Computational Intelligence*, 1(2):31–53, Apr-Jun 2009.

J. Zachman. The Zachman Framework^{TM}. Online, 2010. URL `http://www.zachmaninternational.com/index.php/the-zachman-framework`.

Zeroc. ZeroC Website, Sep 2010. URL `http://www.zeroc.com`.

H. Zuse. *Software Complexity Measures and Models*. de Gruyter and Company, New York, NY, USA, 1990. ISBN 0-89925640-6.